Aplicações mobile híbridas com

Cordova e PhoneGap

SÉRGIO LOPES

SOBRE MIM

Escrevi minha primeira linha de código com 14 anos em 1999, e ela foi em HTML. Daí para CSS e JavaScript foi um pulo. Em seguida, aventurei-me em SSI e PHP, incluindo bancos de dados. Em 2003, iniciei meu curso de Ciência da Computação na USP e nadei em águas mais profundas desde então — Java, C e Python. Cresci bastante em programação back-end.

Mas eu sempre fui apaixonado por front-end.

Leio muito, estudo muito, escrevo muito e programo muito — desde que envolva bastante HTML, CSS e JavaScript. E, de algum tempo para cá, resolvi focar em *mobile*. Estudo e falo muito de Design Responsivo e da Web única como plataforma democrática e universal. Mas sei que há cenários onde Apps podem ser mais interessantes, daí falar de Cordova.

Já trabalhei em algumas empresas, programando em várias linguagens (já até ganhei dinheiro com opensource). Desde 2004, trabalho na **Caelum** como instrutor e desenvolvedor. Foi onde minha carreira decolou e onde mais aprendi, e aprendo todo dia. É onde pretendo passar ainda muitos e muitos anos.

Ensinar e escrever são uma paixão desde o colégio — lembro da

decepção da minha professora de português quando ela descobriu que eu seguiria carreira em exatas. Dar aulas, escrever artigos, bloggar e palestrar são minha maneira de misturar essas habilidades.

Este livro faz parte dessa história. Obrigado por acreditar nele e comprá-lo.

Você pode me encontrar também escrevendo por aí na Web:

- Meu blog pessoal, onde escrevo bastante sobre Web, mobile, front-end em geral: http://sergiolopes.org.
- O blog da Caelum, onde sempre publico artigos sobre front-end: http://blog.caelum.com.br.
- Meu Twitter e meu Facebook onde posto muitos links pra coisas bacanas de front-end e mobile: https://twitter.com/sergio_caelum e https://www.facebook.com/sergio.luis.lopes.jr.
- E também participo de vários fóruns, grupos e listas de discussão de Web, onde nós podemos nos encontrar. Meu favorito é o GUJ (http://www.guj.com.br).

E, se nos toparmos um dia em algum evento, não deixe de me chamar para batermos um papo.

— Sérgio Lopes, 2016

Sumário

Versão: 19.3.5

CAPÍTULO 1

APLICAÇÕES MOBILE COM CORDOVA E PHONEGAP

Ao pensar em construir aplicativos mobile, é importante pensar em quais plataformas atacar. Android domina o mundo dos smartphones no mundo e, principalmente, no Brasil. iOS é bastante usado, ainda mais nas classes sociais mais altas, o que rende usuários de maior poder aquisitivo. Windows Phone é uma boa terceira opção, em franco crescimento. E há ainda BlackBerry, Tizen e outros.

Como desenvolver aplicativos para esse mundo mobile diverso?

As plataformas nativamente oferecem a possibilidade de criar aplicativos. Usando o Android SDK e a linguagem Java, podemos desenvolver para o sistema do Google. A Apple oferece ferramentas para iOS e permite usar Objective-C ou Swift. No Windows Phone, usamos C# e toda suíte de desenvolvimento Microsoft. Cada plataforma tem sua combinação de linguagem e, principalmente, APIs específicas.

A maior parte das plataformas permite usar C++, essencialmente pensando em jogos. Porém, as APIs e as bibliotecas, mudam bastante. Mesmo usando uma linguagem comum, é muito difícil escrever aplicações nativas multiplataforma.

É um problema que não existe na Web. Uma página web bem construída, usando os padrões, é multiplataforma e suporta todos

esses cenários com um só código. HTML, CSS e JavaScript são linguagens padronizadas com APIs padronizadas que funcionam em todo lugar. Mas, não são uma App. Não são instaláveis, não se integram aos recursos avançados de hardware, não expõe recursos dos SDKs nativos. Muitas vezes, é o suficiente, mas e se precisamos de uma App?

1.1 APLICAÇÕES HÍBRIDAS

A solução mais comum atualmente para construção de aplicativos multiplataforma é o **Cordova**, que é basicamente uma mistura. Ele usa o ponto forte da Web de ter linguagens padronizadas e um ambiente de execução, o navegador, para construir aplicativos. São Apps instaláveis que você pode publicar nas lojas, e pode usar recursos nativos da plataforma, mas são escritas em HTML, CSS e JavaScript.

Chamamos de *aplicações híbridas* porque usam as linguagens da Web para construir aplicativos.

Só escrever HTML, CSS e JS não é suficiente para ter um aplicativo no fim. Então, o que o Cordova faz é prover uma **casca nativa** para o nosso aplicativo responsável por subir um browser que fará a execução do nosso código. O papel do Cordova é apenas criar essa janela de navegador para nós, e fazer a comunicação das nossas chamadas de código para chamadas nativas quando necessário.

Essa janela de navegador nativo que vai rodar nosso HTML é comumente chamada de **WebView**. Se você programar em Java no Android ou Objective-C no iOS, vai ver que é relativamente simples criar esse WebView para executar um código HTML. O chato, claro, é fazer isso para diversas plataformas e cuidar das diferenças entre elas.

É aí que entra o Cordova e por isso ele é tão útil. Ele faz todo esse trabalho nativo em diversas plataformas para criar uma WebView e chamar nosso HTML. No fim, ele empacota tudo em um aplicativo - tanto a parte nativa de chamar o WebView quanto todo o nosso código HTML, CSS e JS multiplataforma. O resultado final é um aplicativo específico para cada plataforma que você pode instalar e oferecer nas lojas oficiais.

1.2 O CORDOVA E O PHONEGAP

Falamos bastante do Cordova, mas onde entra o PhoneGap nessa história? **Basicamente, o PhoneGap é uma distribuição proprietária do Cordova.**

O PhoneGap foi criado em 2009 pela empresa Nitobi. Em 2011, a Adobe comprou a empresa mas doou todo o código para o projeto Apache. Nascia aí o Cordova, um projeto opensource tocado pela Apache. O PhoneGap passou a ser o nome do produto da Adobe construído ao redor do Cordova. A base é o Cordova, mas com alguns serviços adicionais da Adobe que podem ser interessantes dependendo do projeto. Veremos mais ao longo do livro.

O ponto é que, se você quer construir aplicações mobile híbridas, pode usar o gratuito e opensource Cordova sem problemas. É o que faremos na maior parte do livro. Algumas ferramentas do PhoneGap podem ser úteis, então é importante conhecê-las também. Mas não são estritamente necessárias.

O Cordova provê a base para uma aplicação híbrida simples. Todos os recursos adicionais e mais avançados são feitos com **plugins**. Há plugins para quase tudo, alguns oficiais da própria Apache ou Adobe, e outros vários feitos por terceiros. No livro, vamos usar vários plugins em diferentes momentos.

1.3 CORDOVA NÃO É WEB

Muito do burburinho que se faz em torno das aplicações híbridas é que são *"o melhor de dois mundos"* ou *"a junção da Web com nativo"*. Eu discordo desse tipo de pensamento.

Uma aplicação híbrida é uma aplicação normal, apenas escrita em linguagens comuns a desenvolvedores Web. **Não é Web**. Isso é importante: Web não é HTML, CSS e JavaScript. Web é uma plataforma universal e aberta de distribuição e navegação de conteúdo. Usamos HTML5 na Web, mas Web não é HTML5.

Quando usamos Cordova, nosso código HTML é empacotado junto à casca nativa para se tornar uma aplicação normal. Há quem chame também de *packaged apps*. E essas Apps são mais próximas de Apps nativas que da Web.

Elas têm as mesmas vantagens e deficiências de Apps normais: precisam ser geradas para cada plataforma, precisam ser disponibilizadas na loja de cada fabricante, e estão submetidas às regras de cada plataforma. Não são navegáveis, não estão na internet, e não têm URLs.

Porém, estão totalmente integradas ao dispositivo. Podem ser instaladas e ser usadas offline. Podem usar APIs da plataforma e usar recursos de hardware avançados. Podem ser divulgados nas lojas e ser vendidas facilmente para os usuários.

PROGRESSIVE WEB APPS

Há muita discussão e alguma implementação já de Web Apps instaláveis com maior integração ao SO (como *Push Notifications* e *Service Workers*). Ainda é bastante incipiente, sem suporte universal, mas certamente um caminho bastante interessante para se observar no futuro próximo.

1.4 APP OU WEB

A questão é que Apps e Web são coisas diferentes. Têm seus pontos fortes e pontos fracos. E a decisão de qual caminho seguir não é trivial. Aliás, é bastante possível que você queira e precise das duas coisas.

A Web favorece a descoberta de conteúdo despretensiosamente: não exige instalação e não exige compromisso do usuário. Ao mesmo tempo, as Apps geram fidelização com seu público e uma experiência mais integrada à plataforma nativa.

Essa discussão é bastante importante e longa. Em meu outro livro, **A Web Mobile**, também da editora Casa do Código, tenho 3 capítulos sobre *Estratégia Mobile*, *Comparativo App e Web* e *O cenário das Packaged Apps*. Recomendo fortemente a leitura se você está refletindo sobre qual caminho seguir.

1.5 QUANDO NATIVO, QUANDO HÍBRIDO?

Aqui, obviamente, seguiremos o caminho das Apps. Mas ainda há uma questão importante: devo fazer aplicativos nativos, ou um híbrido com Cordova?

A decisão é puramente técnica. Do ponto de vista do seu usuário, não há diferença. Um aplicativo híbrido bem construído se integra à plataforma da mesma forma que um nativo. Há diferenças de performance apenas em casos muito específicos que exijam realmente bastante processamento no dispositivo. Nesses casos, escreva nativo.

Nativo é melhor também em Games 3D complicados, ou em aplicações que *precisem* de multithreading. Não é recomendado também para Apps que precisem ficar em background rodando certos serviços. Só o ambiente nativo consegue fazer isso de forma realmente eficiente, sem matar a bateria do usuário.

Como boa parte das Apps não se encaixa nessas categorias, **híbrido é suficiente**. E o que pesa em seu favor é o menor custo de desenvolvimento. Um único código serve todas as plataformas. Não é necessário ter equipes específicas programando Java no Android e Objective-C no iOS, por exemplo. Esse é o principal argumento.

Outro cenário é que se você já tem uma equipe com conhecimentos de HTML, CSS e JS, a curva de aprendizado para o híbrido é bem pequena. Inclusive, é possível aproveitar muita coisa que já foi feita no site Web, se for o caso.

Agora, por mais que o Cordova nos dê acesso a muitos recursos da plataforma, é claro que uma App nativa é quem tem realmente todas as possibilidades a disposição. Se você estiver em uma empresa grande onde diminuir custo de desenvolvimento não é prioridade, onde há equipes especializadas em cada plataforma, pode ser mais interessante construir aplicativos nativos.

Uma preocupação ao desenvolver com Cordova é que a sensação da App não é igual a do nativo. Não usamos componentes da plataforma, mas sim HTML e CSS para criar o visual. E muitas vezes acabamos com um design comum a várias plataformas,

apenas com pequenos ajustes. É claro, uma oportunidade para criar um design único para sua App que extrapole o padrão da plataforma. Mas se você quer algo que use os componentes nativos, aí é melhor uma App nativa.

Pense também em **estratégias mistas**. Você pode começar com uma App híbrida para cobrir rapidamente o maior número de plataformas, e depois ir criando versões específicas nativas quando necessário. O Facebook começou assim, todo em HTML5, e hoje tem aplicações nativas em várias plataformas (além de um excelente site mobile).

1.6 A MORTE DO CORDOVA

É estranho falar, em um livro sobre Cordova, que o Cordova vai morrer. Mas esse é um ponto bem interessante que mostra o real papel do Cordova no mundo. E o ponto é que ele tapa um buraco entre o que os browsers normais podem fazer hoje e o que Apps nativas fazem.

Mas os navegadores têm evoluído e chegado cada vez mais próximos de Apps nativas - vide as novas *Progressive Web Apps*. Nesse sentido, o Cordova vai ficando cada vez menos importante, e um dia talvez seja desnecessário. O próprio criador do PhoneGap disse isso no início do projeto.

Enquanto escrevo o livro no começo de 2016, os navegadores estão começando timidamente a suportar *Service Workers* e *Push Notifications* na Web, e alguns experimentando com Web Apps instaláveis. O Firefox com uma loja onde WebApps podem ser publicadas. O Windows 10 com a mesma ideia. Talvez, no futuro, o cenário Web seja suficiente para muitos tipos de Apps. Até lá, temos o Cordova e o PhoneGap.

1.7 O LIVRO E O PROJETO

O livro é bastante prático. Faremos um projeto mobile real, solucionando vários problemas comuns que você vai passar no dia a dia.

Este não é um guia de referência. O conteúdo está todo espalhado no livro. Vamos aprendendo mais sobre as ferramentas conforme precisarmos no nosso projeto. Se você precisa apenas de uma lista de comandos ou coisa do tipo, é melhor ver a documentação oficial. Aqui, nosso foco é evoluir o conhecimento conforme formos nos aprofundando no projeto.

Veremos os conceitos do Cordova, a arquitetura da plataforma, diversos plugins úteis e mais. É importante saber que o livro foi escrito no começo de 2016. O Cordova está na versão 5.4. Talvez pequenos ajustes sejam necessários se você usar outras versões. Mas tudo deve funcionar por anos da forma que veremos aqui. E eu pretendo atualizar o livro sempre que tivermos mudanças importantes. Consulte a editora Casa do Código para saber se você está lendo a última edição.

O site oficial do Cordova com muito material é:

http://cordova.apache.org.

A documentação oficial da última versão você encontra em:

http://cordova.apache.org/docs/en/latest/index.html.

Capítulo 2

A PRIMEIRA VERSÃO DA APP

2.1 O PROJETO CENOURAPP

Fomos chamados para ajudar a trazer o mundo mobile para a **Só de Cenoura**, uma startup de confeitaria especializada em bolos de cenoura gourmet. Eles ainda funcionam na base do menu clássico em papel e do garçom anotando pedidos no bloquinho. O serviço de delivery é por telefone, também gerando pedidos em papel.

Muita coisa dá errado nesse cenário. Sempre que há mudança de cardápio, é preciso imprimir tudo de novo. A promoção do dia não pode ser colocada no cardápio, uma vez que sempre muda, então fica em uma lousa no canto do restaurante que pouca gente vê. O garçom anota o pedido e passa para cozinha, que muitas vezes não

entende a letra no pedido. O telefone do delivery vive ocupado e os clientes reclamam que não conseguem acompanhar o status do pedido.

Temos de resolver todos esses problemas. **E uma App mobile é a solução pensada**. Daí nasce nosso projeto `CenourApp` que vamos desenvolver.

Nossa estratégia será desenvolver a App por partes, resolvendo um problema por vez. São muitas as possibilidades e muitas coisas para fazer, mas vamos adotar uma estratégia de dar pequenos passos e já colocá-los em uso. Ao longo do livro, vamos evoluindo a App. Mas ao fim de cada capítulo, temos uma App funcional que já pode ser usada na prática no **Só de Cenoura**.

2.2 A PRIMEIRA FUNCIONALIDADE

Nosso primeiro passo é bastante modesto. O pessoal da boleria quer resolver o caso de precisar reimprimir o cardápio a cada semestre quando as opções e preços mudam. Querem também dar um ar mais tecnológico para os clientes e **oferecer o cardápio em um aplicativo mobile**.

Os donos compraram alguns tablets Android baratos que ficarão nas mesas dos clientes. Os cardápios já são desenvolvidos por eles no Photoshop. A diferença é que não queremos mais imprimi-los, e sim mostrar em um aplicativo mobile bonito.

Apesar de todos os tablets da loja hoje serem Android, já queremos desenvolver uma App híbrida pensando em outras plataformas. Se o projeto der certo, e o **Só de Cenoura** crescer, os donos pensam em comprar iPads bonitos para os clientes e um grande tablet touch screen com Windows para deixar na entrada da loja como demonstração do cardápio.

2.3 UMA APP HÍBRIDA É UMA PÁGINA HTML

Nossa App híbrida é apenas um código HTML, CSS e JavaScript empacotado em uma App instalável no Android. Então, podemos começar a desenvolver nosso código HTML.

Essa primeira versão é bem simples. Recebemos dois PNGs com o menu - frente e verso - usado hoje no restaurante. Nossa App precisa mostrar essas imagens e dar uma forma simples de navegação para o usuário trocar de página.

As imagens com os 2 menus são `<img>` simples. Podemos fazer o menu de troca com *radio buttons* e seus respectivos *labels*. Um HTML simples:

```
<html>
<head>
    <meta name="viewport"
          content="width=device-width,initial-scale=1.0">
    <link rel="stylesheet" href="menu.css">
</head>
<body>

    <input type="radio" name="opcao" id="opcao-bolos" checked>
    <label for="opcao-bolos">Bolos</label>

    <input type="radio" name="opcao" id="opcao-bebidas">
    <label for="opcao-bebidas">Bebidas</label>

    <img src="imagens/menu-bolos.png"
         id="menu-bolos" class="menu">

    <img src="imagens/menu-bebidas.png"
         id="menu-bebidas" class="menu">

</body>
</html>
```

Crie esse conteúdo HTML em um arquivo `index.html` na pasta de sua escolha.

VIEWPORT

Repare que usamos a meta tag `viewport` como nos sites mobile e responsivos comuns. Para saber mais sobre o funcionamento dos `viewport` s na Web, consulte o capítulo 10 do meu livro **A Web Mobile**.

Usando CSS, é possível exibir apenas a imagem que estiver selecionada utilizando seletores avançados do CSS3. Com a pseudoclasse `:checked` , sabemos qual opção está selecionada. E com o seletor de irmão adjacentes `~` , selecionamos a foto correspondente. Uma forma de fazer isso é:

```
#opcao-bolos:checked ~ #menu-bebidas,
#opcao-bebidas:checked ~ #menu-bolos {
    display: none;
}
```

Com esse código, se uma certa opção estiver marcada, a foto da *outra* opção ficará escondida. Há muitas maneiras de implementar essa funcionalidade. Essa versão simples com CSS nos é suficiente para a primeira versão da App.

Ainda podemos melhorar bastante o estilo. A imagem do menu pode estourar em certas telas menores, então queremos configurar sua largura máxima. E vamos esconder os *input radio*, para usar apenas o `label` para escolha de opção:

```
input[type=radio] {
    display: none;
}
.menu {
    width: 100%;
}
```

Nosso próximo passo é deixar os labels com cara de botões,

colocar uns ícones bonitos e acertas outras coisas decorativas:

```
body {
    background: #3D1A11;
    font-family: sans-serif;
    margin: 0;
    text-align: center;
}

label {
    background: center 0.5em no-repeat #563429;
    background-size: 4em;
    color: white;
    display: block;
    font-size: 75%;
    padding: 4em 0 1em;
    text-transform: uppercase;
}
    label[for=opcao-bolos] {
        background-image: url(imagens/icone-bolos.svg);
    }
    label[for=opcao-bebidas] {
        background-image: url(imagens/icone-bebidas.svg);
    }
    :checked + label {
        background-color: #E4876D;
    }
```

Por fim, podemos posicionar os botões embaixo na tela fixamente e lado a lado:

```
label {
    width: 50%;

    position: fixed;
    bottom: 0;
    z-index: 1;
}
    label[for=opcao-bolos] {
        left: 0;
    }
    label[for=opcao-bebidas] {
        right: 0;
    }

.menu {
    margin-bottom: 100px;
```

}

Escrevendo esse HTML e CSS, junto com os arquivos das imagens, já teremos um projeto funcionando no browser. Você pode abrir no navegador, diminuir a janela e testar como se fosse um dispositivo móvel. Isso ajuda muito no desenvolvimento. Mas, claro, ainda não é uma App. Faremos isso no capítulo seguinte.

Figura 2.2: Screenshot das telas do nosso aplicativo

Código completo do exemplo

O código completo, com todo HTML, CSS e as imagens, você encontra no GitHub:

https://github.com/sergiolopes/cenourapp.

> **BUG NO ANDROID VELHO**
>
> Versões antigas do WebKit não se davam bem com a pseudoclasse `:checked` e seletores adjacentes `~`. Se for o seu caso, existe um hack que resolve esse problema que eu deixei no final deste CSS:
>
> https://github.com/sergiolopes/cenourapp/blob/master/menu.css#L84-L89.
>
> Mais para a frente, veremos como resolver todos os problemas de compatibilidade do Android com o Crosswalk.

2.4 EFEITOS NO HTML COM CSS

Nosso foco será, claro, construir aplicativos com Cordova e PhoneGap. Não vou focar tanto na parte HTML/CSS das Apps. O recado importante é que escrevemos HTML, CSS e JavaScript **normais**, que todo desenvolvedor front-end está careca de escrever. Não há segredo.

Neste nosso exemplo mesmo, ainda simples, se quisermos adicionar uns efeitos, é só usar CSS. Por exemplo, fazer uma transição entre as duas telas, escorregando da direita para a esquerda. Com *CSS Transitions*, isso é bem simples.

Edite a parte das imagens e coloque um novo `div` ao redor delas, que será responsável por deslizar quando a transição acontecer:

```
<div class="container-menus">
    <img src="imagens/menu-bolos.png" alt="Nossos bolos"
         id="menu-bolos" class="menu">
```

```
    <img src="imagens/menu-bebidas.png" alt="As Bebidas"
         id="menu-bebidas" class="menu">
</div>
```

E agora o CSS. O deslocamento em si é feito com *CSS transform*, fazendo um `translateX` para deslocar horizontalmente. A vantagem do `transform` é que ele é rápido, principalmente em mobile, por ser resolvido na GPU e ter aceleração de hardware. A animação é feita com a propriedade `transition` , onde controlamos tempo e tipo de efeito.

Mais alguns ajustes são necessários. Vamos posicionar fora da tela a imagem não selecionada, para depois animá-la para dentro quando selecionar. Para evitar um scroll lateral, precisamos de um `overflow-x:hidden` .

```
html,
body {
    overflow-x: hidden;
    width: 100%;
}
.container-menus {
    transform: translateX(0);
    transition: transform 300ms ease;
    width: 200%;
}

.container-menus .menu {
    float: left;
    width: 50%;
}

#opcao-bebidas:checked ~ .container-menus {
    transform: translateX(-50%);
}
```

Prefixos transform e transition

Dependendo do navegador, pode ser necessário ainda colocar alguns prefixos nas propriedades. Todos os modernos não precisam, mas talvez você precise fazer coisas como:

```
-webkit-transform: translateX(0);
-webkit-transition: -webkit-transform 300ms ease;
```

Implemente essas duas mudanças e veja o efeito sendo aplicado. E pense em muitos outros efeitos possíveis, como, por exemplo, animar os botões. O ponto aqui é entender que tudo aquilo que fazemos com HTML, CSS e JavaScript na Web podemos fazer nas nossas aplicações híbridas também.

CAPÍTULO 3

ADOBE PHONEGAP BUILD

Quem faz o trabalho pesado das Apps híbridas é o Apache Cordova. Vamos estudá-lo a fundo a partir do próximo capítulo. Mas o Adobe PhoneGap tem coisas bem interessantes, principalmente no início de um novo projeto, por isso vamos falar um pouco dele antes.

3.1 A DIFERENÇA ENTRE CORDOVA E PHONEGAP

O PhoneGap é 90% Cordova. Ele é uma *distribuição* do Cordova, contém o Cordova dentro de si e permite fazer tudo o que o outro faz. Mas o **PhoneGap tem serviços a mais**, coisas que a Adobe colocou que facilitam o desenvolvimento, principalmente no início.

Atualmente, são 3 serviços principais:

- **PhoneGap Desktop** - App Desktop simples para criar e servir Apps PhoneGap;
- **PhoneGap Developer App** - App mobile que permite testar sua App no celular sem precisar empacotar nem instalar;
- **PhoneGap Build** - Serviço de build remoto, que permite gerar a App final sem você precisar da infraestrutura na sua máquina.

Veremos um pouco de cada um deles, começando pelo *Build*.

3.2 O QUE É O PHONEGAP BUILD

Quando começarmos a usar Cordova, você verá que uma dificuldade bem grande é arrumar o ambiente de desenvolvimento. Há muita coisa para instalar e configurar. Apesar de ser multiplataforma, a arquitetura do Cordova/PhoneGap exije a geração de uma App nativa no final, aquela casca que abre a *WebView*.

Isso quer dizer que **você precisa dos SDKs nativos de cada plataforma** para usar Cordova. Você não precisa conhecer Java nem escrever código Android, mas se quiser um instalável Android (apk) no final, vai precisar instalar o *Android SDK*. A mesma coisa serve para o iOS, o Windows Phone e as demais plataformas.

Entraremos nos detalhes dessas configurações nos capítulos seguintes. A única coisa chata é que começar com Cordova é muito complicado. Fizemos nosso HTML e CSS no capítulo anterior e só queríamos uma Appzinha para vermos no celular. Nada muito complexo, mas com Cordova, não dá.

Entra aí o **PhoneGap Build**. Este é um serviço cloud da Adobe que tem toda a infraestrutura de Android, iOS e Windows Phone instalada para você. Isso quer dizer que você só escreve seu HTML/CSS/JS, manda para a nuvem, e recebe de volta uma App nativa em cada plataforma - um *apk* no Android, por exemplo.

3.3 USANDO PHONEGAP BUILD

O *Build* é um serviço pago da Adobe, mas com uma camada gratuita. Se você tem a assinatura da *Creative Cloud*, tem acesso já ao plano máximo do *Build*; se não, custa US$10 por mês. No plano

gratuito, temos quase todas as vantagens do plano pago, apenas que as Apps não podem ser muito grandes e estamos limitamos a *apenas uma* App privada - mas infinitas Apps públicas.

A forma mais simples de subir nosso código para o PhoneGap Build é criar um arquivo ZIP com todo o HTML, CSS, JS e imagens necessários. O arquivo raiz *precisa* ser um `index.html` . Então, abra a pasta onde você criou o exercício do capítulo anterior e faça um ZIP com todo seu conteúdo.

Se preferir, o código do primeiro exemplo está no meu GitHub e você pode clicar direto em *Download ZIP*:

https://github.com/sergiolopes/cenourapp.

Agora, vamos subir o ZIP no *PhoneGap Build*. Para nossos testes, o plano gratuito é suficiente. Entre em http://build.phonegap.com e crie uma conta *Free*. Você deve precisar de um usuário AdobeID.

Ao entrar na primeira vez, ele deixa você criar Apps públicas a partir de repositórios públicos do GitHub, ou Apps privadas com repositório privado ou upload de ZIP. Vamos usar essa última opção do ZIP. Marque a aba *Private*, e clique em *Upload a .zip file*. Selecione o arquivo ZIP com nosso projeto.

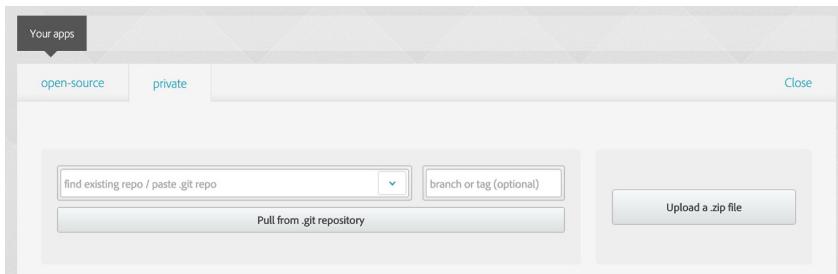



Figura 3.1: Upload de zip privado para o Build

3.4 UMA APP ANDROID PRONTA PARA RODAR

Feito o upload, clique para fazer o build. Ele vai demorar um pouco e logo dará a opção de baixar as Apps finais geradas. Por padrão, Android e Windows Phone.

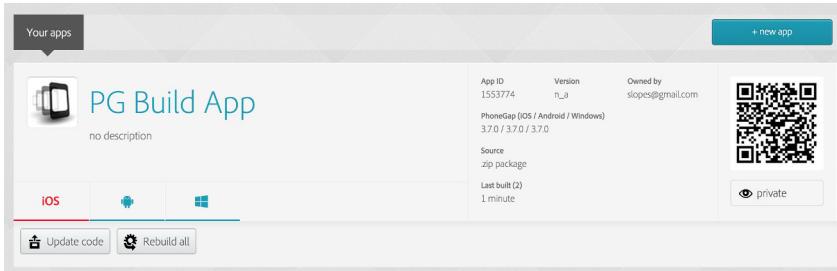
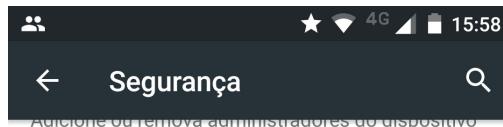



Figura 3.2: Após o build

Se clicar no ícone do Android, ele vai baixar o `apk`, um arquivo instalável do Android. Ou, ainda mais fácil, escaneie o QRcode direto no aparelho para baixar. É só instalar.

Instalação em um aparelho Android

Seu aparelho precisa estar aberto para instalação de arquivos de fora da Play Store. Se você nunca fez uma instalação fora da loja, vai precisar marcar uma opção.

Vá nas *Configurações* e em *Segurança*. Lá, marque a opção *Permitir Instalação de Fontes desconhecidas*.

Feito isso, basta clicar no `apk` que você baixou e seguir a instalação. Ele vai pedir umas permissões, finalizar a instalação e dar a opção de já abrir a App.

E o Windows Phone e o iOS?

O *Build* consegue gerar arquivos para Windows Phone e iOS também. Inclusive a versão Windows já é gerada automaticamente. Porém, é mais complicado do que parece. Bem mais. Vamos ver com calma essas plataformas depois, mas um resumo é:

- **Windows Phone**: para instalar o arquivo gerado em um aparelho, ele precisa estar registrado. Para isso, precisamos de um computador Windows com SDK instalado e uma conta (paga) de Developer da Microsoft.
- **iOS**: para buildar, precisamos de certificados específicos da Apple. E, para isso, temos de ter uma conta (paga) Developer da Apple. E ter um Mac com Xcode para gerar as chaves localmente.

Dos 3, o Android é a **plataforma mais amigável para desenvolvedores**. Podemos buildar de forma simples e instalar em qualquer aparelho, sem burocracia.

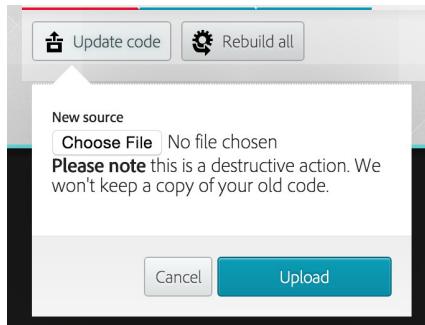
Por enquanto então, vamos focar no Android. Mas veremos mais à frente como suportar iOS e Windows Phone também.

3.5 CONFIGURAÇÕES DO PHONEGAP BUILD

Repare que a App que subimos ganhou um nome e um ícone padrões. É porque subimos apenas o HTML/CSS simples. Aí o *Build* gera essas configurações padrões.

Mas podemos inserir esses e outros parâmetros avançados em um **XML de configuração do PhoneGap Build**. É bem simples. Um arquivo `config.xml` na raiz do projeto, mesmo nível do `index.html`. Ele segue a estrutura de XML do *W3C Widgets*.

Para colocar um nome na App, uma descrição e informações do autor, o arquivo seria:

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://www.w3.org/ns/widgets"
        xmlns:gap="http://phonegap.com/ns/1.0"
        id="org.sergiolopes.cenourapp"
        version="1.0.0">

    <name>
        Só de Cenoura
    </name>

    <description>
        Menu do restaurante Só de Cenoura.
    </description>

    <author href="http://sergiolopes.org"
            email="qualquer@sergiolopes.org">
        Sérgio Lopes
    </author>
</widget>
```

Repare que usamos as tags `<name>`, `<description>` e `<author>`. Também repare que na declaração da tag `<widget>` passamos um `id` único - geralmente baseado no nome do domínio ao contrário - e a versão da aplicação.

Subindo uma nova versão da App para o Build

Com o arquivo XML criado, basta recriar o ZIP da aplicação e subir novamente no *Build*. Lá na nossa App, deve ter um botão **Update Code** onde você manda um novo ZIP.

Se preferir, deixei esse código pronto em uma branch separada no GitHub. De lá, você pode baixar o ZIP dessa versão:

https://github.com/sergiolopes/cenourapp/tree/build-config.

Preferências

Além das opções básicas do XML, é possível configurar várias preferências. Por exemplo, travar a orientação com:

```
<preference name="orientation" value="portrait" />
```

É uma configuração interessante para nossa App já que o formato do menu foi pensado para ser usado apenas no modo retrato. Podemos ainda indicar que nossa App deve rodar em modo tela cheia, sem a barra de status no topo.

```
<preference name="fullscreen" value="true" />
```

Outra opção simples é configurar as permissões da App. Por padrão, o *Build* gera uma App que pede muitas permissões, justo pensando em uma App com muitas capacidades. Como nossa App não é nada especial ainda, podemos usar:

```
<preference name="permissions" value="none" />
```

É interessante para não afugentar o usuário com muitas permissões.

Pensando em Android, podemos ainda restringir a versão dos usuários. O *Build* por padrão suporta desde o Android 2.1, que é muito velho e tem uma WebView bastante bugada. Podemos restringir para Android 4.1 e superiores, que são a maioria no Brasil hoje. Basta usar a versão do SDK (no caso, 16):

```
<preference name="android-minSdkVersion" value="16" />
```

Repare que estamos lidando apenas com Android agora, mas o *Build* tenta gerar iOS e Windows Phone. Podemos restringir isso para facilitar e pedir apenas o build do Android:

```
<gap:platform name="android" />
```

Uma lista completa de preferências você encontra em:

http://docs.build.phonegap.com/en_US/configuring_preferences.md.html.

> **REINSTALAR APP**
>
> Dependendo das mudanças, o *Build* pode gerar um `apk` que o Android rejeita como update (por causa de versões, assinatura e outros motivos). Nesses casos, **desinstale a versão antiga primeiro** antes de instalar a nova.

3.6 ÍCONE E SPLASH SCREEN

Duas configurações melhoram bastante a experiência do usuário: um bom ícone para ficar na home dele, e uma bela splash

screen para aparecer enquanto a App carrega.

O ícone deve ser um arquivo PNG quadrado. É possível fazer diversos tamanhos pensando em plataformas e telas diferentes. Ou apenas um arquivo relativamente grande, que configuramos com:

```
<icon src="icon.png" />
```

A splash screen é também um PNG e pode ter vários tamanhos. Pensando só em Android por enquanto, um arquivo de 720x1280 é suficiente. Ele é configurado com:

```
<gap:splash src="splash.png" />
```

Deixei um projeto com dois arquivos configurados para você usar aqui:

https://github.com/sergiolopes/cenourapp/tree/build-config-completo.

3.7 RODANDO EM MODO FIXO NO ANDROID

Um recurso importante para aplicativos como esse que vão rodar no restaurante é de restringir o acesso dos usuários a demais recursos do aparelho. Queremos deixar nossa App rodando em tela cheia, e impossibilitar o usuário de sair da App e acessar outras coisas no aparelho.

A partir do Android 5, é muito fácil fazer isso. Para habilitar o recurso, vá nas *Configurações* do aparelho e selecione o menu *Segurança*. Lá para o final, entre na opção *Fixação de tela*, e marque para ativar. Você ainda pode marcar *Pedir PIN antes de desafixar*, um recurso interessante para não deixar o usuário sair da App se não souber a senha do aparelho.

Aí abrimos a nossa App no aparelho. Com ela aberta, aperte o botão **Recentes** da barra de navegação do Android, aquele botão de

alternar os aplicativos recentes. Selecione o aplicativo atual, dê um scroll para cima e verá um ícone de **Fixar App** (*Pin App*) parecido com uma tachinha.

Ao selecionar essa opção, o aplicativo é fixado e o usuário não pode sair dele sem saber a senha do aparelho.

Para saber mais sobre esse recurso, consulte:

https://support.google.com/nexus/answer/6118421?hl=en.

3.8 INDO ALÉM COM PHONEGAP BUILD

O *Build* tem outros serviços que podem ser interessantes em certos casos.

- **PhoneGap CLI**: acesso ao *Build* por linha de comando para não precisar fazer upload manual no site;
- **Uso de plugins**: todo plugin Cordova ou PhoneGap direto na nuvem;
- **Hydration**: update automático do código HTML das Apps usando servidor da Adobe sem o cliente reinstalar a App;
- **Debug remoto**: com Weinre, direto nos servidores da Adobe;
- **Assinatura dos binários**: para distribuição final nas lojas.

Na documentação deles, você pode ver mais sobre esses serviços:

http://docs.build.phonegap.com/.

Vamos ver vários desses recursos durante o livro aplicados ao Cordova - como CLI, debug, distribuição na loja e uso de plugins diversos.

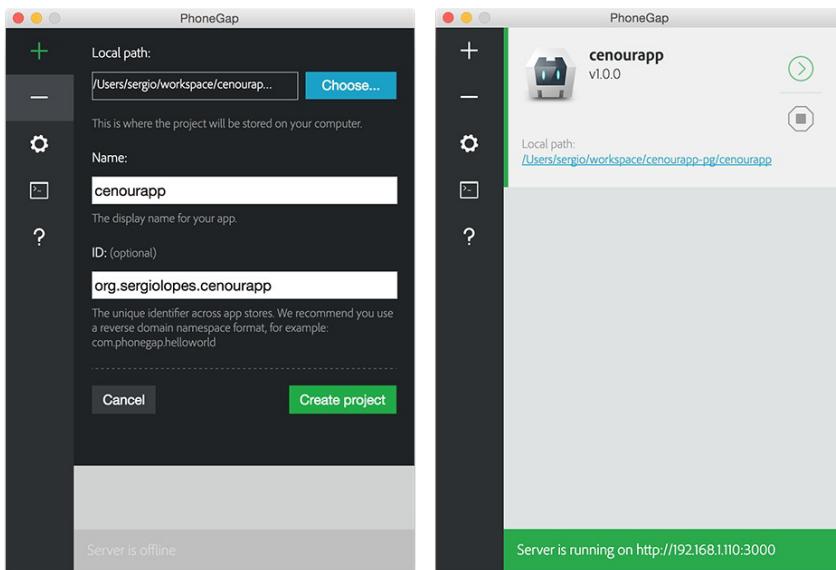
Capítulo 4

PHONEGAP APP

O PhoneGap oferece mais um serviço muito útil para testes, principalmente na fase inicial do projeto, onde queremos ver rapidamente um protótipo ou uma aplicação simples. É o **PhoneGap Developer App**. Usar o *Build* toda hora durante o desenvolvimento não é muito prático, já que você precisa regerar e reinstalar a aplicação a cada mudança.

O **PhoneGap Developer App** é uma App que você instala no aparelho e *linka* com o projeto na sua máquina. Ele automaticamente puxa a versão mais recente do código e abre como se fosse a App final. Ou seja, você instala uma vez a App do PhoneGap e pode usá-la para carregar e recarregar o código de todas as suas Apps. Facilita muito o desenvolvimento.

4.1 PHONEGAP DESKTOP APP

Antes de abrir o código no celular, precisamos *instalar o PhoneGap no computador*. Há duas formas de fazer isso: via linha de comando e com uma aplicação Desktop. Vamos fazer a mais fácil.

O *PhoneGap Desktop App* é uma aplicação **Desktop** que permite criar Apps PhoneGap de forma muito fácil. Ela não possui todas as ferramentas, mas é bem simples e suficiente para o que precisamos agora.

Enquanto escrevo, ela ainda é beta e disponível só para Mac e

Windows. Baixe direto no GitHub e instale:

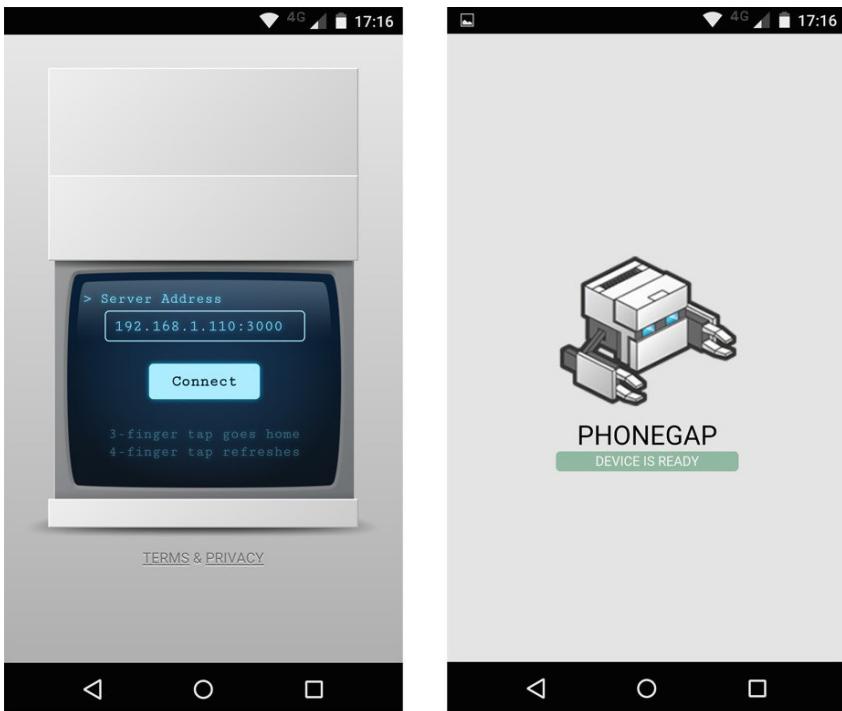
https://github.com/phonegap/phonegap-app-desktop/releases.

Abra o programa e clique no + para criar um novo projeto. Dê uma pasta, um nome e um pacote quaisquer.

Repare que, no final, ele sobe um endereço que aparece embaixo da janela - um IP na porta 3000. Anote esse endereço, pois é o que vamos usar para conectar os aparelhos móveis.

4.2 UM PROJETO PHONEGAP

Abra a pasta onde você mandou gerar o projeto e veja seu conteúdo. Há muitos arquivos e subpastas lá dentro - `hooks` , `platforms` , `plugins` , `www` . Quando entrarmos em profundidade no Cordova, discutiremos em detalhes o papel de cada parte.

Por enquanto, o importante para nós é saber que **todo nosso**

código vai dentro da pasta `www` . Todo nosso HTML, CSS, JavaScript e imagens devem ser colocados lá dentro. Inclusive, se abrir a pasta agora, você verá que o PhoneGap gerou já um `index.html` e até certas imagens e estilos de uma App simples padrão - um *HelloWorld*.

Pegue nosso código anterior do menu do *Só de Cenoura* e jogue dentro da pasta `www` deste novo projeto PhoneGap. Sobrescreva os arquivos que vieram por padrão com os nossos, do nosso projeto.

4.3 PHONEGAP DEVELOPER APP

Próximo passo é instalar a *PhoneGap Developer App* no dispositivo. Há versões para Android, iOS e Windows Phone. É só baixar nas lojas oficiais mesmo. Os links você encontra aqui: http://app.phonegap.com.

Instalada a App, abra e veja que ela pede um endereço. É o endereço que a *Desktop App* nos deu no passo anterior, então é só digitar lá e apertar *Connect*. Ele vai baixar o código da nossa máquina e carregar no aparelho. Excelente para testar rapidamente!

Importante

O aparelho e o computador precisam estar na mesma rede para se encontrarem.

Outra dica: você pode tocar a tela com 3 dedos para voltar a home, ou tocar com 4 dedos para forçar um reload.

Por fim, esse recurso sobe um serviço de *livereload* também. Isso significa que você pode editar seu código e ele será recarregado automaticamente no dispositivo.

CAPÍTULO 5

UM AMBIENTE REAL PARA TRABALHAR COM APPS HÍBRIDAS

5.1 POR QUE USAR SÓ O PHONEGAP NÃO VAI TE LEVAR MUITO LONGE

Usamos os serviços mais famosos do PhoneGap, o *Build* e a *App*. Eles nos ajudaram a testar rapidamente no dispositivo e a ganhar produtividade nesse início da nossa App. Mas eles deixam buracos importantes no dia a dia, e é isso que precisamos discutir aqui.

PhoneGap App não é tão útil assim

O *PhoneGap App* ajuda a visualizar a App, porém ele tem limitações. Você não vê a instalação nem carregamento da App (*splash screen*); seu código é exibido meio que magicamente. Ele também não tem suporte a todos os plugins. Eles até tentam e suportam muitos plugins oficiais do Cordova e PhoneGap, mas você provavelmente vai acabar usando algum plugin externo alguma hora.

Em Apps avançadas, você também acabará precisando customizar alguma coisa nativa, como por exemplo, uma configuração do iOS no XCode. E não vai conseguir rodar isso na

PhoneGap App. Na prática, nada substitui o **teste da aplicação real** no aparelho e nos emuladores.

O PhoneGap Build não substitui o ambiente local

Já o *PhoneGap Build* nos ajudou a obter um `apk` para Android com relativa facilidade. Seria possível até subir nossas chaves de desenvolvedor e obter o `apk` final para subir na *Play Store*. O sistema Android está muito bem coberto, na verdade, o problema são as demais plataformas.

O *Build* até consegue nos dar um arquivo `xap` do Windows Phone, mas você não consegue instalá-lo no aparelho diretamente para testar. O aparelho precisa estar registrado e, para isso, você precisa de uma máquina Windows com o SDK do Windows Phone instalado. Ou seja, se você for ter todo o trabalho de configurar o SDK do Windows, pode muito bem fazer seus builds locais, não precisa do *PhoneGap Build*.

No iOS, a situação também é estranha. O *Build* não consegue gerar o arquivo `iap` se você não subir as suas chaves pessoais de desenvolvedor. É possível gerar as chaves, subir no *PhoneGap Build* e aí instalar a App no seu device pessoal para testes. Para isso, ou você faz uma assinatura de Developer da Apple (US$99 anuais), ou precisa de um Mac com XCode 7 instalado.

Se você já tem o Mac com XCode, você consegue gerar as chaves localmente, exportá-las e depois importá-las no *Phonegap Build*. Assim, ele gerará o arquivo `iap` que você pode instalar no aparelho usando iTunes. Mas, se você já tem um Mac, pode simplesmente dar um *Play* na App e ela abre no aparelho, sem precisar da conta paga para testar (só depois, para publicar na loja).

E você vai precisar de emuladores

Lembre-se que para isso estamos assumindo que você possui um dispositivo real para testes - um iOS, um Windows Phone, um Android. Recomendo fortemente que você tenha esses aparelhos. Nada substitui um teste real. Mas, na prática, você **vai precisar de emuladores**, seja porque não tem o aparelho daquela plataforma, seja porque quer testar uma versão diferente do sistema. Ou mesmo porque você não quer ficar rodando no aparelho toda hora.

Se você quer emuladores locais das plataformas, você precisará instalar todo o ambiente, então será necessário um Mac com XCode para emular iOS. Não tem jeito. Assim como você precisa de um Windows com Visual Studio e Windows Phone SDK, e do Android SDK e seus emuladores - que, ainda bem, funcionam no Windows, Mac e Linux.

Além de desenvolver e testar Apps, o ambiente local é essencial para depois **publicar a App na loja**. Ou seja, na prática, você vai precisar instalar o ambiente de desenvolvimento de cada plataforma cedo ou tarde. O *Build* ajuda muito a começar, mas logo você precisará do seu próprio ambiente.

5.2 UM AMBIENTE LOCAL PARRUDO

Cada plataforma tem suas exigências. Precisamos basicamente de quatro coisas: desenvolver, emular, testar no aparelho e publicar na loja.

Vamos resumi-las:

- **Android**: mais amigável de todos, disponível para Windows, Mac, Linux.
    - *Desenvolvimento:* Android SDK;
    - *Emuladores:* vêm com o SDK, são bem lentos (alternativa: *Genymotion*);

- *Teste em aparelho:* qualquer aparelho;
    - *Publicar*: exige uma conta de desenvolvedor, custa 25 dólares uma vez.

- **Windows Phone**: exige um Windows 8.1 ou Windows 10, na versão Pro 64 bits.

    - *Desenvolvimento:* Visual Studio Express, gratuito;
    - *Emuladores:* vêm com o Visual Studio, mas exigem um processador Intel moderno com suporte a Hyper-V, e são bem pesados para rodar;
    - *Teste em aparelho:* precisa registrar o aparelho no SDK e instalar App pelo Desktop;
    - *Publicar*: via Web, exige uma conta de desenvolvedor, custa 19 dólares por ano.

- **iOS**: exige um Mac.

    - *Desenvolvimento:* XCode, gratuito;
    - *Emuladores:* vêm com XCode, muito bons e rápidos;
    - *Teste em aparelho:* a partir do XCode 7, pode testar direto no seu aparelho sem conta paga;
    - *Publicar*: exige XCode e uma conta de desenvolvedor que custa 99 dólares anuais.

Durante o livro, nós vamos configurar e usar as 3 plataformas. Vamos desenvolver com elas, testar, emular e, no fim, publicar nas respectivas lojas. O Android, por ser o mais fácil, vai ser nosso primeiro alvo, mas vamos evoluindo.

5.3 SOU USUÁRIO DE WINDOWS, NÃO TENHO UM MAC!

Você consegue desenvolver Android e Windows Phone tranquilamente. Mas se precisar de iOS, temos um problema.

É absolutamente impossível desenvolver para iOS sem ter um Mac. Nós precisamos buildar o projeto, gerar as chaves de desenvolvedor, usar emulador do iOS, executar no aparelho de testes e publicar na App Store da Apple. Tudo isso para uma App simples, normal. Se não, você vai precisar customizar algo nativo (código Objective-C).

Tudo isso exige Mac, mas para algumas coisas dá para nos virarmos sem um:

- **Buildar:** é possível usar o PhoneGap Build para isso, mas é necessário gerar as chaves do desenvolvedor em um Mac;
- **Geração das chaves:** com alguns hacks, é possível hoje em dia gerar as chaves no Windows se tiver a conta paga da Apple – com a conta gratuita para testes, apenas no Mac;
- **Testar no dispositivo:** se usar o PhoneGap Build com suas chaves, é possível instalar no aparelho usando iTunes, porém, com Mac, é só dar um play no XCode;
- **Emulador:** *não tem jeito, precisamos de um Mac;*
- **Customizar código nativo:** *não tem jeito, precisamos de um Mac;*
- **Publicar na loja:** *não tem jeito, precisamos de um Mac.*

Então, você precisa de um Mac, e temos tais opções:

1. **Comprar um Mac.** Apesar de serem conhecidos pelo valor mais alto, é possível achar bons negócios em máquinas usadas. Ou podemos comprar um Mac mini, que é mais barato que os demais.
2. **Montar um Hackintosh.** É tentar instalar o OSX em um PC normal. É uma tarefa nada fácil, pois há várias limitações de hardware. Além de não ser permitido pela Apple.
3. **Hackintosh virtualizado.** Tentar o OSX em um VirtualBox ou

VMWare na sua máquina. Também é bastante complicado e não permitido, mas há guias por aí.

4. **Alugar um Mac na nuvem.** Várias empresas oferecem Mac no cloud e você paga por hora de uso. Solução bem barata e fácil. É mais lento, claro, mas é uma opção bastante viável, principalmente se seu objetivo for apenas gerar as chaves de desenvolvimento uma vez para usar no *PhoneGap Build*.

Já vá pensando nisso desde o começo do projeto. Se não, no fim, você não vai conseguir entregar o projeto no iOS. O melhor é comprar um Mac mesmo. Se você estiver em uma empresa, já solicite a compra no início do projeto.

Minha máquina pessoal é um Mac, então, na verdade, eu passo pelo próximo problema:

5.4 SOU USUÁRIO DE MAC, NÃO TENHO UM PC!

Você consegue desenvolver Android e iOS sem problemas. Se precisar de Windows Phone, temos um problema. Você precisa de uma máquina Windows, e não um qualquer, uma versão Pro, 64 bits, em um hardware com processador Intel com suporte a Hyper-V (a maioria dos Macs modernos tem esse tipo de processador). Como fazer então?

O *PhoneGap Build* permite buildar a App na nuvem. Ele ajuda bastante se você quer só publicar. Você consegue buildar e subir o arquivo na loja via Web, já assinado. Mas, para desenvolver, ele não ajuda. Para rodar o arquivo no aparelho real, você precisa de Windows. Fora que você também precisa de Windows para registrar seu aparelho, customizar código nativo e rodar o emulador.

Então, você vai precisar comprar um PC ou instalar Windows

no seu Mac. Há duas formas para fazermos isso.

BootCamp

Você particiona seu HD e instala o Windows separado do Mac. Quando ligar a máquina, você pode escolher bootar no Windows e usá-lo em toda sua plenitude. Esse tipo de setup é tranquilo, gratuito, rápido e suportado tanto pela Apple quanto pela Microsoft. Você só precisa comprar uma licença do Windows Pro, claro.

O ponto ruim é precisar reiniciar a máquina toda hora para trocar de Windows para Mac, e vice-versa.

Virtualização

Costuma ser a solução mais usada. Tanto **Parallels** quanto **VMWare Fusion** rodam muito bem. O essencial é que suportam *virtualização aninhada*, para poder rodar o emulador do Windows Phone.

Ambos são produtos pagos e você ainda precisa pagar a licença do Windows também. O *VirtualBox* é uma solução gratuita de virtualização e suportará bem quase tudo, menos o emulador. Isso porque ele ainda não consegue rodar a virtualização aninhada da forma que o Windows Phone precisa. Todo o resto - buildar, testar no device, publicar na loja, editar código nativo - funciona no VirtualBox.

5.5 SOU USUÁRIO LINUX, O QUE FAÇO?

Basicamente, você só consegue desenvolver para Android. Se quiser iOS, vai precisar de Mac. Se quiser Windows Phone, vai precisar de Windows. Veja as duas seções anteriores, pois você

precisará se virar com OSX e Windows.

CAPÍTULO 6

PREPARANDO CORDOVA E ANDROID

Neste capítulo, vamos preparar o ambiente para desenvolvimento com Cordova e, a princípio, com Android. Como nossa primeira App só precisa rodar em Android, começaremos por ele. É um bom jeito de começar também por ser mais simples de configurar do que o iOS e o Windows Phone. Mais para a frente, veremos como adicionar essas outras plataformas.

A instalação no Windows e no Mac é praticamente idêntica. Nos exemplos, usei Windows 8.1, mas o processo deve ser mais ou menos o mesmo desde o Windows 7 até o 10. No Mac, usei Yosemite.

Se precisar de ajuda, recomendo fortemente o fórum do GUJ em http://www.guj.com.br.

> **INSTALANDO NO LINUX**
>
> Se você usa Linux, você deve ser capaz de instalar tudo deste capítulo também. O processo depende de qual distribuição você está usando, então é preciso pesquisar um pouco. Em geral, porém, a ideia é usar o gerenciador de pacotes da sua distribuição e instalar as coisas.

Do que nós precisamos

Essencialmente, vamos instalar:

- Node.js
- Cordova
- Java JDK
- Android SDK
- VirtualBox
- Emuladores Genymotion
- Google Chrome

Se você já tiver esses componentes instalados, pode pular este capítulo. Se não, acompanhe a seção específica de cada um.

6.1 NODE.JS

Precisamos do Node para usar o Cordova. Entre no site do Node em http://nodejs.org, e clique no grande botão **Install**. Ele vai baixar o pacote e você pode simplesmente executá-lo.

Faça a instalação padrão com *Next*, *Next*, *Finish*, e deixando as opções padrões marcadas.

Ao fim, para saber se a instalação foi bem sucedida, abra o **terminal** e digite:

```
node -v
```

Ele deve confirmar a versão instalada do node.

ABRINDO O PROMPT NO WINDOWS

Vamos usar bastante o terminal para desenvolver cordova. Para abri-lo, você pode ir ao menu *Iniciar* e achar a categoria *Windows System*, onde há o ícone para o *Command Prompt*. Outra opção é abrir a janela de execução com *Win+R* e digitar `cmd` . Chamarei de **Terminal**.

ABRINDO O TERMINAL NO MAC

No Mac, você acha o **Terminal** em *Applications/Utilities*. Ou, se preferir, procure por **Terminal** no Spotlight.

6.2 CORDOVA

O Cordova é um pacote do Node.js que vamos instalar com sua ferramenta de pacotes, chamada `npm` .

No Windows, feche e abra novamente o terminal, e digite:

```
npm -g install cordova
```

No Mac e no Linux, precisamos do `sudo` para instalar globalmente, então faça:

```
sudo npm -g install cordova
```

Deve demorar um pouco para fazer a instalação completa para nós.

6.3 JAVA SDK

Precisamos do Java para usar o Android SDK. Se você não tem certeza se possui o Java instalado, abra um terminal e rode o comando:

```
javac -version
```

Precisamos do Java na versão 1.8.x ou superior. Para instalá-lo, acesse:

http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Localize o pacote **Java Platform JDK** na versão 8 ou superior (não precisa baixar a versão Netbeans). Baixe e siga a instalação padrão com *Next*, *Next*, *Finish*.

Ao final, abra um novo terminal e teste novamente: `javac -version` .

6.4 ANDROID SDK

Apesar de ser um projeto híbrido, o Cordova exige as ferramentas nativas instaladas para montar a App final e testá-la. Por isso, vamos precisar do **Android SDK** que funciona no Windows, Mac e Linux.

Entre no site do Android e baixe o SDK para sua plataforma:

https://developer.android.com/sdk/index.html#Other.

No Windows, siga a instalação padrão com *Next*, *Next* e *Finish*. No final, tem a opção *Start SDK Manager*, deixe-a marcada. Deverá abrir o *SDK Manager* do Android. Se não, vá ao menu *Iniciar* e ache o *SDK Manager*.

Já para Mac e Linux, o pacote nada mais é que um ZIP que você pode descompactar onde quiser, como a home do seu usuário ou

alguma outra pasta. Faça essa descompactação e *anote o caminho completo onde colocou.*

Você vai precisar adicionar o caminho no `PATH`. Para isso, abra o arquivo `~/.bashrc` e adicione a seguinte linha, colocando o caminho correto na sua máquina:

```
export PATH=$PATH:<caminho_android_sdk>/tools:<caminho_android_sdk
>/platform-tools:<caminho_android_sdk>/build-tools
```

E no terminal recarregue essas novas configurações com:

```
source ~/.bashrc
```

Por fim, você precisa abrir o *SDK Manager* rodando no terminal:

```
android sdk
```

ANDROID STUDIO

Se você pretende se aprofundar no desenvolvimento Android, recomendo instalar o Android Studio completo. Aqui, para nós, o Android SDK é suficiente, até porque usaremos emuladores Genymotion e o próprio Cordova para gerenciar a parte Android da nossa App.

Configuração do Android SDK

Instalado o Android SDK, precisamos então adicionar certos componentes a ele. Abra o **Android SDK Manager** como no passo anterior. Vamos instalar:

- SDK Platform
- Android SDK Platform-tools
- Android SDK Build-tools

Marque as últimas versões do *Build-tools*, *Platform-tools* e do *SDK Platform*. Clique em *Install*. Ele baixará bastante coisa e é bem demorado para instalar.

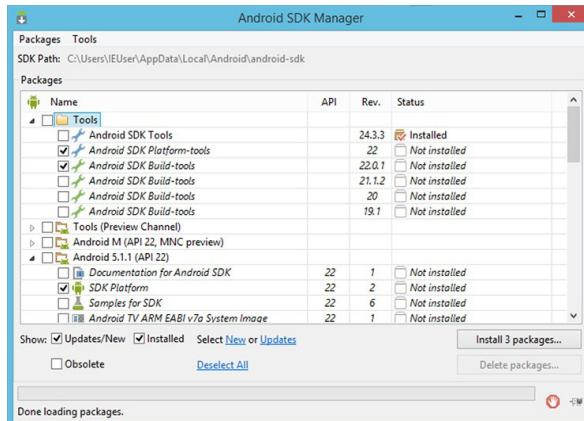



Figura 6.1: Instalação das ferramentas no Android SDK Manager

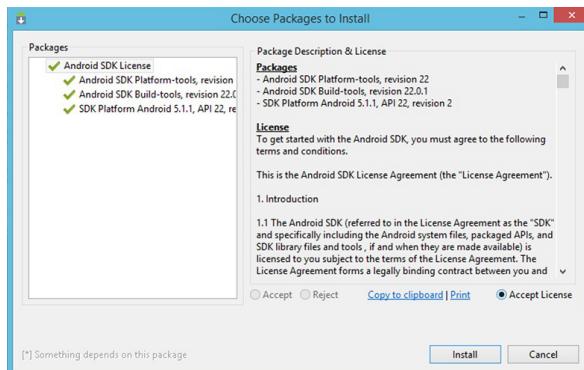



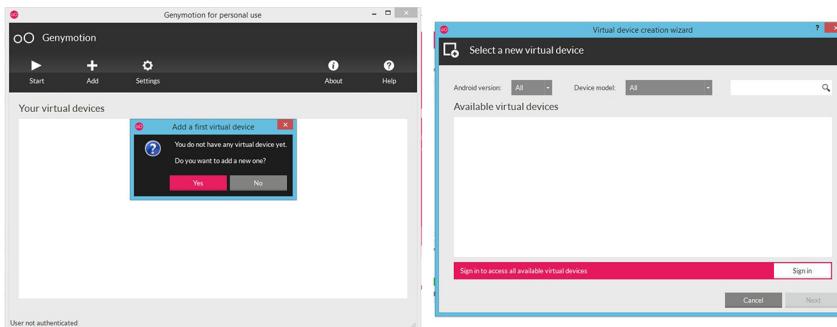
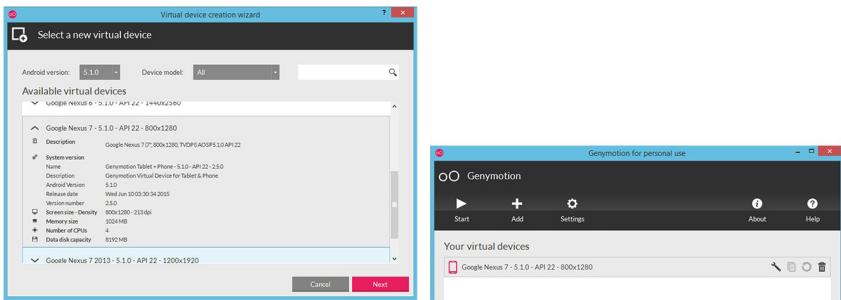
Figura 6.2: Opções a instalar no Android SDK Manager

6.5 GENYMOTION

O último passo é instalar um emulador de Android. O SDK traz emuladores oficiais, mas eles são *muito lentos*. Uma ferramenta que cresceu bastante no mercado é o **Genymotion**, que permite rodar o Android direto virtualizado no VirtualBox. E ele é **muito mais**

rápido.

O Genymotion é gratuito para uso pessoal e você pode baixá-lo em: https://www.genymotion.com/.

Clique em *Download* e selecione a versão *Free*. Você vai precisar criar uma conta, mas é rápido.

Na versão Windows, você tem a opção de baixar o Genymotion já com VirtualBox embutido (ou só Genymotion caso já tenha o VirtualBox instalado). Baixe e siga a instalação padrão.

No Mac e Linux, é necessário instalar o VirtualBox separadamente. Entre em http://virtualbox.org, baixe o arquivo e siga a instalação padrão. Você talvez precise reiniciar a máquina após a instalação do VirtualBox para que o Genymotion o encontre.

Baixando emulador no Genymotion

Instalado o Genymotion e o VirtualBox, precisamos baixar emuladores de versões específicas do Android que nos interessam.

Abra o **Genymotion**. Ele pedirá seu usuário e senha, que você criou no Site. Depois, você pode adicionar novos emuladores. Teste instalar um **tablet com Android 5**, por exemplo, ou outra versão.

Para testar, clique em *Start* e veja se o emulador abre e funciona.

Você pode instalar outros emuladores também, em mais versões e tamanhos de tela.

Capítulo 7

COMEÇANDO UMA APP NO ANDROID

A App do menu do **Só de Cenoura** já está sendo usada no restaurante e os clientes estão gostando. Agora, o pessoal pediu para desenvolvermos uma nova App, para ajudar as garçonetes a anotarem os pedidos.

Essa nova app se chama `garconapp` e, a princípio, será pensada apenas para smartphones Android, que são o que as garçonetes usam no restaurante.

7.1 PREPARAÇÃO DO AMBIENTE CORDOVA E ANDROID

Antes de prosseguir, esteja certo de já ter configurado seu ambiente para desenvolvimento Cordova, pelo menos com Android. Vamos usar aqui:

- Cordova CLI
- Android SDK
- Emuladores Genymotion ou um dispositivo Android real
- Google Chrome no Desktop

Veja o capítulo anterior para um passo a passo detalhado de preparação do ambiente Android.

7.2 USANDO CORDOVA PELA PRIMEIRA VEZ

A linha de comando do Cordova vai nos dar acesso a tudo que precisamos para criar e gerenciar projetos de Apps híbridas. O primeiro passo é **criar o projeto**.

Para isso, usamos o comando `cordova create` que recebe 3 argumentos importantes:

- Pasta da App
- Identificador (nome de pacote)
- Título da App

Abra um terminal e entre na pasta onde quer criar o novo projeto. E aí digite em um só comando:

```
cordova create garconapp org.sergiolopes.garconapp "Garçonete Só d
e Cenoura"
```

IDENTIFICADOR PRECISA SER ÚNICO

O argumento com nome do pacote precisa ser único para você se pretende publicar a App um dia. Não use o exemplo que dei com meu domínio, pois esse endereço já está sendo usado.

Ele deve gerar a pasta `garconapp` cheio de arquivos dentro. Aproveite e abra essa pasta para ter uma ideia. Para nós agora, **apenas uma pasta importa**, a `www` . Dentro dela, vamos colocar o HTML, CSS, JavaScript e imagens da App. Repare até que o gerador colocou coisas lá já. É só um *HelloWorld** simples.

Mais à frente, vamos nos aprofundar em cada uma das outras pastas do projeto.

7.3 PLATAFORMAS

O Cordova suporta diversas plataformas - Android, iOS, Windows Phone, Blackberry e mais. Por padrão, não suporta nenhuma. Criado o projeto, precisamos **adicionar suporte às plataformas** que queremos.

Antes de adicionarmos o Android, nossa plataforma final, vamos usar uma outra, mais simples e fácil de usar, e muito útil para testes. É a plataforma *do próprio navegador*. Como nosso projeto é um HTML, ele pode ser aberto em um navegador comum. E isso ajuda muito no desenvolvimento.

Abra um terminal e entre na pasta do projeto. Então, adicione a plataforma `browser` com:

```
cd garconapp
cordova platform add browser
```

Assim, podemos executar o projeto que tem aquele HTML simples de *HelloWorld* na plataforma `browser`. É bem simples:

```
cordova run browser
```

Ele vai abrir uma nova instância do Google Chrome com algumas configurações de segurança alteradas, e mostrará a App padrão do Cordova, que tem mais ou menos essa cara:

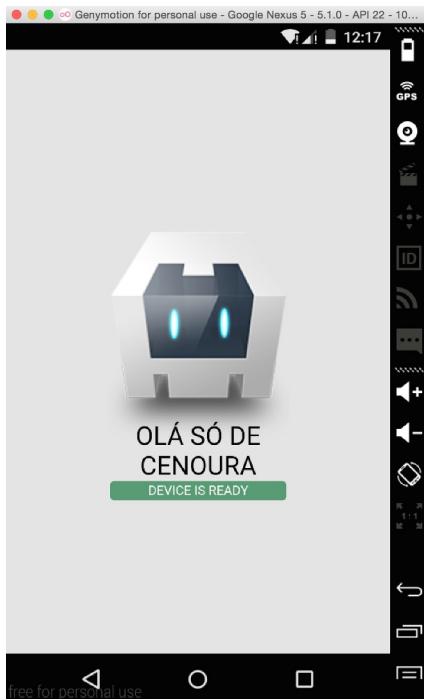
Agora faça um teste: abra o arquivo `index.html` na pasta `www`, e edite alguma frase do HTML - por exemplo, o `<h1>`. Então

rode `cordova run browser` novamente para ver a mudança no navegador.

7.4 RODANDO NO ANDROID

Se o ambiente estiver propriamente configurado para Android, executar nesta plataforma é tão simples quanto no *browser*. Primeiro, adicionamos a plataforma:

```
cordova platform add android
```

E, para rodar:

```
cordova run android
```

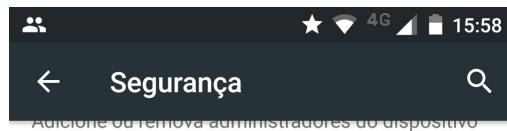
Isso provavelmente deu um erro para você. Ele reclama que não há dispositivo Android nenhum. Logo, precisamos de um dispositivo Android para rodar, seja real ou emulado. Vamos usar os **emuladores Genymotion** que instalamos antes.

Abra o *Genymotion* e inicie algum emulador. Depois de iniciado, tente novamente:

```
cordova run android
```

Pronto, agora a App deve executar no emulador.

7.5 EXECUTANDO NO APARELHO ANDROID

Se você tiver um aparelho com Android, podemos rodar a App com a mesma facilidade. O aparelho precisa estar preparado para duas coisas:

- Instalação de aplicativos fora da loja
- Aberto para depuração USB

Vamos fazer as duas coisas.

Configuração de fontes desconhecidas

Se ainda não fez, vá às *Configurações* do Android e entre na seção *Segurança*. Lá, **habilite a opção *Fontes Desconhecidas***, que permite a instalação de aplicativos fora da Play Store. O nome do

menu pode variar um pouco de acordo com o fabricante, mas todos têm essa opção.

Depuração USB

Esse processo permite o controle do aparelho via USB. Assim, ao plugar o cabo no Desktop, o Cordova - via o Android SDK - pode rodar nossas Apps direto no aparelho, sem burocracia.

Primeiro, você precisa que o menu *Programador* ou *Desenvolvedor* esteja habilitado nas *Configurações* do Android. Ele é o penúltimo menu, logo antes de *Sobre*, e não vem habilitado por padrão. Para habilitá-lo, vá ao menu *Sobre* e localize o *Número da versão*. Toque 7 vezes no menu e ele mostrará uma mensagem de que abriu as opções de desenvolvimento.

Volte às *Configurações* do Android. Agora você deve ver a opção *Programador*.

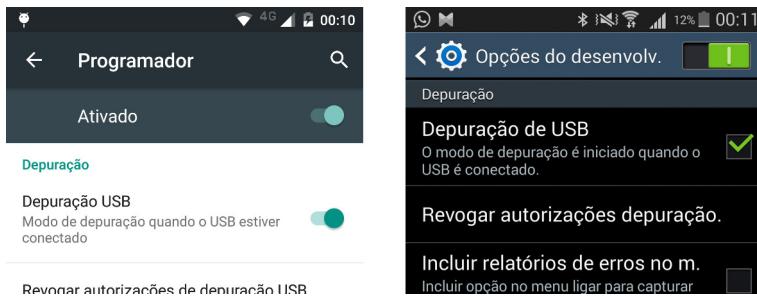



Figura 7.4: Opção em um Moto X e em um Samsung Galaxy Note

Entre nessa opção e habilite o menu **Depuração USB**. Conecte o cabo USB no Desktop e ele deve pedir para confirmar a chave única do computador (marque a opção de *sempre permitir*).

No Windows

No Windows, na primeira vez que você conecta o aparelho por USB, ele faz a **instalação dos drivers**. Esse processo demora um pouco, mas é importantíssimo, se não o PC não reconhece o aparelho. No Mac, isso já não é necessário.

Na maioria dos aparelhos, a instalação dos drivers é *automática* quando plugamos a primeira vez. Mas pode ser que você precise instalar os drivers *manualmente* caso o Android SDK não consiga conectar no aparelho. Neste caso, baixe o driver do seu fabricante nesta lista:

http://developer.android.com/tools/extras/oem-usb.html.

Tudo configurado! Para ter certeza de que o celular está pronto para rodar Apps Cordova pela linha de comando, digite no terminal `adb devices` e ele deve listar seu aparelho com um código do lado.

Finalmente executando no aparelho

Toda essa configuração é necessária apenas na primeira vez em cada aparelho. Feito tudo, basta plugar o cabo USB e rodar no terminal:

```
cordova run android
```

Você deve ver a App demo do Cordova abrindo no aparelho:

7.6 GERANDO UM APK PARA DISTRIBUIÇÃO

Quando acabarmos de desenvolver a App, vamos querer instalá-la nos aparelhos das garçonetes do restaurante, e não vamos ficar conectando via USB e rodando linha de comando para isso, para cada aparelho.

Precisamos **gerar uma App Android instalável**. No Android, isso é um arquivo `apk` que pode ser distribuído para as pessoas e instalado em qualquer aparelho. Quando o Cordova executa no nosso aparelho de testes, na verdade ele gera esse `apk`, instala no aparelho e abre a App.

Podemos gerar o **apk** com:

```
cordova build android
```

Ele gera um arquivo na pasta `platforms/android/build/outputs/apk/android-debug.apk`.

Este pode ser instalado nos aparelhos, mas repare que ele é de *debug*, o que significa que ele é aberto para desenvolvedores, ou seja, não é o aplicativo final.

Podemos gerar o **apk final** com:

```
cordova build android --release
```

Agora ele gera um `platforms/android/build/outputs/apk/android-release-unsigned.apk` . O nome *unsigned* significa que não está assinado digitalmente, o que não é importante agora para nós.

Você pode mandar esse arquivo por e-mail para as pessoas, por exemplo, e todo mundo pode instalar no próprio aparelho. A única configuração necessária é que o aparelho esteja com a opção de *Fontes Desconhecidas* habilitada (não precisa da *depuração USB*).

Esse processo é o de distribuição interna para pessoas que você controla. Para distribuir via **Play Store** para o mundo todo, vamos precisar ainda de mais algumas coisas que veremos mais para a frente.

Para agora, já conseguimos instalar nossa App facilmente em todos os aparelhos do Restaurante. O próximo passo é implementar a `garconapp` .

Capítulo 8

MATERIAL DESIGN

Precisamos implementar a `garconapp`, a App que as garçonetes do *Só de Cenoura* vão usar para coletar os pedidos dos clientes. É uma App de uso interno e com foco inicial apenas em Android. Vamos usar o *Material Design* para isso.

> Este capítulo e o próximo mostram como montar uma App em HTML. Não mostram nada novo do Cordova em si, apenas HTML e CSS. Sinta-se à vontade para pulá-los se quiser ver apenas Cordova estritamente. No final do próximo capítulo, você pode baixar o código completo do HTML/CSS para usar nos exemplos de Cordova e seguir o livro.

8.1 O DESIGN DO GOOGLE

Junto com o lançamento do Android 5 em 2014, o Google anunciou uma **nova linguagem visual** para seus sistemas e Apps. O **Material Design** é um estilo visual *flat*, mas que usa efeitos e sombras sutis para melhorar a experiência. O Google diz que se baseou no *papel* e como ele funciona no mundo real.

Se você usou alguma App do Google recentemente, já viu esse estilo. A maior parte das Apps e todo o Android foram atualizados ao longo do tempo. Até na Web.

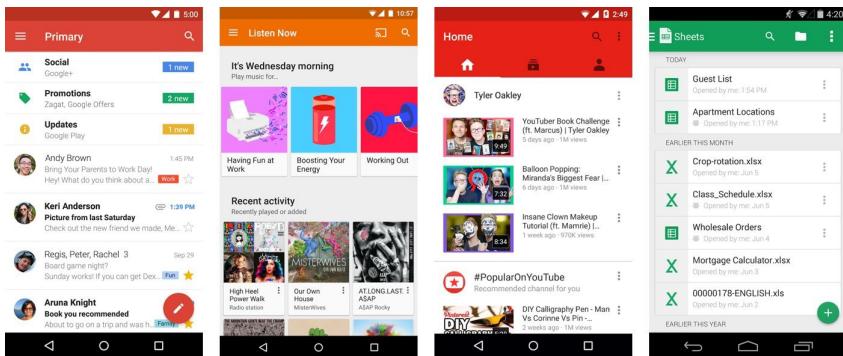



Figura 8.1: Gmail, Play Music, Youtube, Sheets

Existe uma espécie de especificação que explica os conceitos e aplicações do *Material Design* (http://www.google.com/design/spec/material-design). É bem interessante e didática. Assim, usaremos o *Material Design* na `garconapp`.

Implementando o Material Design

Sendo uma linguagem visual, qualquer tecnologia ou plataforma pode usar *Material Design*. Agora nos interessa como aplicar o *Material* em HTML e CSS, tanto na Web quanto em Apps Cordova. Existem várias soluções possíveis, desde frameworks CSS simples até soluções completas de MVC ou componentes.

Hoje, as mais famosas são:

- Polymer Paper Elements
- Angular Material
- Material Design Lite
- Materialize
- Material UI (React)
- MUI CSS

Há vantagens e desvantagens em cada framework, e recomendo

que você pesquise e teste para saber qual o melhor para o seu projeto. Aqui, vamos usar o **Materialize**, porque ele é bem simples de usar apenas com HTML, possui uma vasta biblioteca de componentes e atende com facilidade o cenário da nossa App.

8.2 A GARÇONAPP

A App não tem grandes exigências de layout e design, mas precisa ser bastante focada e útil para as garçonetes do restaurante. Ela precisa:

- Listar todas as opções de bolos e bebidas;
- Anotar os itens do pedido;
- Indicar qual a mesa que fez o pedido;
- Despachar o pedido para a cozinha automaticamente.

A cara final da App vai ser algo como:

8.3 O MATERIALIZE

O primeiro passo é jogar fora o código que o Cordova gerou para nós no projeto `garconapp`, dentro da pasta `www`. Entre na pasta, e apague o `index.html` e as subpastas de `css`, `js` e `img`.

As dependências

Agora baixe o Materialize no site oficial (http://materializecss.com), descompacte o ZIP e copie as pastas `css`, `js` e `font` para dentro da `www` do nosso projeto.

Baixe também o jQuery em http://jquery.com, pois o Materialize precisa dele para os componentes avançados. Jogue o arquivo `jquery.min.js` na pasta `js`, dentro de `www`.

Precisamos também dos ícones oficiais do Material Design do Google. Eles oferecem uma *icon font* pelo serviço Google Fonts, o que é muito fácil de usar, mas exige que o usuário esteja online. Para nós, seria interessante baixar os ícones offline e embuti-los na App. É um pouco trabalhoso, pois exige copiar o CSS que o Google indica aqui:

http://google.github.io/material-design-icons/#icon-font-for-the-web.

Para facilitar, deixei já preparado os ícones e o CSS, em:

https://github.com/sergiolopes/garconapp/tree/master/www/icons.

Ainda dentro da pasta `www`, crie um novo arquivo `index.html`. Ele deve seguir o esqueleto padrão de um HTML5 com viewport mobile, e já importando o CSS e o JS do Materialize:

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta name="viewport"
          content="width=device-width, initial-scale=1">
    <meta charset="utf-8">
    <title>Só de Cenoura Garçom</title>
    <link rel="stylesheet" href="icons/material.css">
    <link rel="stylesheet" href="css/materialize.min.css">
</head>
<body>


    <script src="js/jquery.min.js"></script>
    <script src="js/materialize.min.js"></script>
</body>
</html>
```

PROJETO INICIAL

Se preferir, deixei esse esqueleto com Materialize, jQuery e Material Icons já pronto no GitHub. É só clicar em *Download ZIP* e partir daí. É só entrar em

https://github.com/sergiolopes/garconapp.

8.4 ITENS EM ABAS

Nossa App precisa mostrar uma aba com uma lista de opções de *Bolos* e outra com *Bebidas*. No *Materialize*, as abas são compostas por 2 componentes. Há uma lista (`ul`) de abas com links para o conteúdo de cada aba; e há o conteúdo das abas em si, que nada mais é que um `div` com certo `id` .

A lista de abas tem classe `tabs` , e cada aba é uma `tab` :

```
<ul class="tabs">
  <li class="tab"><a href="#bolos">Bolos</a></li>
  <li class="tab"><a href="#bebidas">Bebidas</a></li>
</ul>
```

Repare em como há links para IDs com `#bolos` e `#bebidas`. Precisamos definir `div` s com esses IDs para serem o conteúdo de cada aba. E cada aba deve ter uma lista de itens, que já criaremos com a classe `collection` e vários itens com `collection-item`:

```
<div id="bolos" class="section">
    <div class="collection">
        <a class="collection-item">Só de Cenoura</a>
        <a class="collection-item">Com Nutella</a>
        <a class="collection-item">De Brigadeiro</a>
        <a class="collection-item">Açucarado</a>
    </div>
</div>

<div id="bebidas" class="section">
    <div class="collection">
        <a class="collection-item">Espresso</a>
        <a class="collection-item">Capuccino</a>
        <a class="collection-item">Mocachino</a>
    </div>
</div>
```

Teste esse HTML no navegador e você terá algo como:

Figura 8.3: Abas com conteúdo em listas

Ainda não estamos nos preocupando com cores e customizações de design.

Mais conteúdo nas abas

Você pode acrescentar mais itens nas abas, claro, para bater com

o cardápio completo que tínhamos na primeira App no começo do livro. Uma dica é criar várias `collection` e separá-las com subtítulos separadores (use classe `container` para deixar alinhado).

Por exemplo, a aba de bebidas poderia ser:

```
<div id="bebidas" class="section">
    <h6 class="container">Cafés</h6>
    <div class="collection">
        <a class="collection-item">Espresso</a>
        <a class="collection-item">Capuccino</a>
        <a class="collection-item">Mocachino</a>
    </div>

    <h6 class="container">Refrigerantes</h6>
    <div class="collection">
        <a class="collection-item">Soda</a>
        <a class="collection-item">Guaraná</a>
        <a class="collection-item">Coca</a>
    </div>
</div>
```

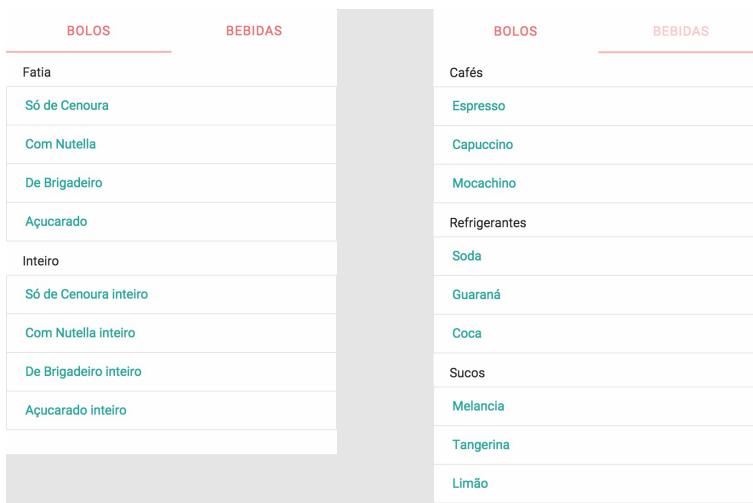



Figura 8.4: Sugestão de mais itens nas abas

8.5 EFEITOS DE ONDAS DO MATERIAL

DESIGN

Um dos efeitos mais famosos do *Material Design* é o *ripple effect* ou *waves effect*. Nos itens clicáveis, ele faz um efeito circular animado ao clicar/tocar. Com *Materialize*, é bem fácil adicionar esse efeito.

Usamos a classe `wave-effect` nos elementos, e ele já põe um efeito padrão. Você pode mudar também o tipo. Por exemplo, `wave-light` faz uma onda branca em vez de escura. Veja mais em http://materializecss.com/waves.html.

Podemos adicionar nos itens da lista. Por exemplo:

```
<div class="collection">
    <a class="collection-item waves-effect">Soda</a>
    <a class="collection-item waves-effect">Guaraná</a>
    <a class="collection-item waves-effect">Coca</a>
</div>
```

8.6 CUSTOMIZAÇÃO DO VISUAL DO MATERIALIZE

As cores padrão do *Materialize* seguem uma palheta inspirada no *Material Design*, mas elas podem ser trocadas. Há muitas cores já predefinidas em classes CSS:

http://materializecss.com/color.html.

Para deixar os itens da lista em preto, podemos fazer:

```
<a class="collection-item waves-effect black-text">Com Nutella</a>
```

Os títulos divisores das categorias poderiam ser marrons com:

```
<h6 class="container brown-text">Sucos</h6>
```

As abas queremos com um laranja e texto em branco. Eu escolhi uma variação que envolve duas classes para o laranja, `yellow`

`darken-4` . E já aproveitamos e colocamos o efeito de *waves* nas abas também.

O HTML das abas pode ser **modificado** para:

```
<ul class="tabs yellow darken-4">
    <li class="tab">
        <a href="#bolos"
           class="white-text waves-effect waves-light">
            Bolos
        </a>
    </li>
    <li class="tab">
        <a href="#bebidas"
           class="white-text waves-effect waves-light">
            Bebidas
        </a>
    </li>
</ul>
```

CSS próprio

Ao testar, você reparará que, nesta versão do *Materialize*, as linhas debaixo das abas não mudam de cor para branco. Vamos precisar de CSS manual para isso. O bom é que aproveitamos e configuramos algumas outras coisas.

Crie um arquivo `css/estilos.css` e referencie no HTML *após* o `materialize.css` :

```
<link rel="stylesheet" href="css/estilos.css">
```

Para deixar a linha das abas em branco, coloque no CSS:

```
.tabs .indicator {
    background-color: white;
}
```

Aproveite também e mude o fundo da App para um cinza claro bem neutro:

```
body {
    background-color: #F2F2F2;
```

```
}
```

Neste ponto, nossa App já tem uma cara bem bonita e customizada para a **Só de Cenoura**:

8.7 TOPO COM TÍTULO E ÍCONES

O topo da App deve ter uma barra de navegação com o título da App e o ícone do menu. O topo com um título à esquerda e um ícone à direita, alinhados pelo centro, ficaria:

```
<div class="valign-wrapper yellow darken-4 white-text">
    <h5 class="titulo">Só de Cenoura</h5>

    <div>
        <i class="material-icons waves-effect
                  waves-light waves-circle">more_vert</i>
    </div>
</div>
```

A classe `valign-wrapper` alinha o título e o ícone verticalmente no centro. A classe `material-icons` aplica um ícone com nome `more_vert` usando a fonte oficial do Google.

Repare que já colocamos a cor de fundo e dos textos e também o

efeito de *waves* no ícone. E, ainda com a classe `waves-circle` , o efeito é circular, algo comum em ícones soltos no *Material Design*.

Se rodar agora no navegador, quase tudo já funciona. Só o título ainda é meio feio e desalinhado. Por isso, colocamos a classe `titulo` , que é nossa, para poder estilizar no CSS.

```
.titulo {
    font-size: 1.3rem;
    margin-left: 5%;
    margin-right: auto;
}
```

Ao rodar no navegador, temos agora:

Topo fixo

Mais um efeito bastante comum nas Apps é o título e a barra de navegação (abas) ficarem fixos no topo. E ainda, com *Material Design*, terem um efeito de profundidade para parecerem sobrepostos ao restante do conteúdo.

No HTML, vamos criar um `div` que **envolva a barra do topo e as abas**. Vou dar uma classe `topo-fixo` a ele e também a classe do *Materialize* `z-depth-2` , que adiciona o efeito de profundidade.

```
<div class="topo-fixo z-depth-1">
```

Esse `div` vai envolver o `div` com classe `valign-wrapper` , e

a `ul` com classe `tabs`.

No CSS, nosso topo fixo é um mero `position:fixed`:

```
.topo-fixo {
    position: fixed;
    top: 0;
    width: 100%;
    z-index: 2;
}
```

Por fim, um ajuste nos conteúdos das abas para não ficarem por trás do topo fixo:

```
.section {
    padding-top: 125px;
}
```

Teste novamente no navegador. Repare em como um HTML e um CSS simples já nos trazem um resultado bem bacana com cara de App.

> O código final dos exemplos deste capítulo você encontra nesta branch no GitHub:
>
> https://github.com/sergiolopes/garconapp/tree/design.

CAPÍTULO 9

COMPONENTES RICOS NA APP

Dando seguimento à App do capítulo anterior, precisamos agora implementar as funcionalidades reais da App. Isso envolve um menu de navegação, uma lista de itens selecionados, uma janela de confirmação e mais.

Para essas funcionalidades, vamos usar também JavaScript, além do HTML e do CSS.

9.1 ANOTANDO O PEDIDO

Temos já uma lista de produtos que o cliente pode pedir. Queremos que a garçonete possa tocar nos itens da lista e ir adicionando elementos no pedido. No *Materialize*, podemos colocar um contador de itens com a classe `badge` . Veja um exemplo:

```
<a class="collection-item waves-effect black-text">
    Guaraná
    <span class="badge brown-text">2</span>
</a>
```

Ou seja, dentro do `collection-item` , podemos colocar um `badge` com o número já adicionado. Mas não podemos simplesmente colocar no HTML como vimos anteriormente, precisamos **criar dinamicamente** esses elementos e ir incrementando conforme a garçonete toca.

Badge dinâmica com jQuery

Vamos usar JavaScript para essa funcionalidade. Criaremos um evento de `click` nos `collection-item`:

```
$('.collection').on('click', '.collection-item', function(){
});
```

Dentro do evento, precisamos criar um `span.badge` se ainda não existir um. Seria algo como:

```
var $badge = $('.badge', this);
if ($badge.length === 0) {
    $badge = $('<span class="badge brown-text">0</span>')
                .appendTo(this);
}
```

Repare que começa com valor zero. A ideia é ir incrementando esse valor a cada clique, algo bem simples:

```
$badge.text(parseInt($badge.text()) + 1);
```

O código completo ficaria então:

```
$('.collection')
    .on('click', '.collection-item', function(){

        var $badge = $('.badge', this);
        if ($badge.length === 0) {
            $badge = $('<span class="badge brown-text">0</span>')
                        .appendTo(this);
        }

        $badge.text(parseInt($badge.text()) + 1);
    })
```

Crie um arquivo `js/app.js` e coloque esse código anterior. Não se esqueça de importar o arquivo no final da página após os outros scripts:

```
<script src="js/app.js"></script>
```

Teste clicar nos itens da lista e veja os números incrementando.

Cafés	
Espresso	3
Capuccino	1
Mocachino	
Refrigerantes	
Soda	2

Toast

Um componente legal do *Material Design* é aquela barra escura de notificação que aparece quando uma ação acontece. Isso é chamado de **Toast**. Com *Materialize*, é bem fácil de acionar um, basta chamar `Materialize.toast()` .

Na nossa App, podemos mostrar um toast no evento de click que fizemos para mostrar qual produto foi adicionado:

```
var nomeProduto = this.firstChild.textContent;
Materialize.toast(nomeProduto + ' adicionado', 1000);
```

9.2 FLOATING ACTION BUTTON

Um padrão bastante comum em Apps *Material Design* é o uso do botão de ação flutuante no canto direito inferior. A maioria das Apps do próprio Google tem um botão desses.

Com *Materialize*, podemos criar um com a classe `fixed-action-btn` . O botão em si é um `btn-floating` e você pode controlar tamanho, *waves* e cores como nos outros elementos:

```
<div class="fixed-action-btn">
```

```
    <a href="#confirmacao" id="confirmar"
    class="btn-floating btn-large waves-effect waves-light brown">
        <i class="material-icons">done</i>
    </a>
</div>
```

Repare que o link aponta para uma seção `#confirmacao`, que ainda não temos na App. A ideia é criar um **modal de confirmação**, e esse será o ID do modal.

9.3 MODAL

O *Materialize* simplifica bastante o uso de modais nas nossas Apps, mas ainda temos de fazer certas configurações. Primeiro, é necessário realizar a estrutura do modal.

Esta nada mais é que um `div` com classe `modal` que possui dois filhos: um elemento com classe `modal-content` e outro com `modal-footer`. O importante é que usemos o ID que o botão está referenciando - `confirmacao` no nosso caso.

```
<div id="confirmacao" class="modal modal-fixed-footer">
    <div class="modal-content">
        ...
    </div>
    <div class="modal-footer">
        ...
    </div>
</div>
```

Para o nosso botão realmente acionar o modal, duas configurações são importantes. Primeiro, no link que dispara, precisamos **adicionar a classe** `modal-trigger`. No nosso caso, é no `<a>` que possui a classe `btn-floating`.

Agora é preciso inicializar o plugin dos modais no código JavaScript. Acrescente no arquivo `js/app.js` a inicialização:

```
$('.modal-trigger').leanModal();
```

Se você testar agora, já vai ver um modal vazio abrindo ao apertar o botão de ação flutuante. Vamos ver seu conteúdo.

O conteúdo principal

O modal tem uma função bem simples: mostrar um resumo do pedido e perguntar qual a mesa. Vamos implementar essas coisas com componentes do *Materialize*. Dentro do `div` com classe `modal-content` , adicionaremos:

- Um título simples com:

```
<h5>Resumo do pedido</h5>
```

- Um `input` que pergunta o número da mesa:

```
<input id="numero-mesa" type="number"
       class="validate" placeholder="Número da mesa">
```

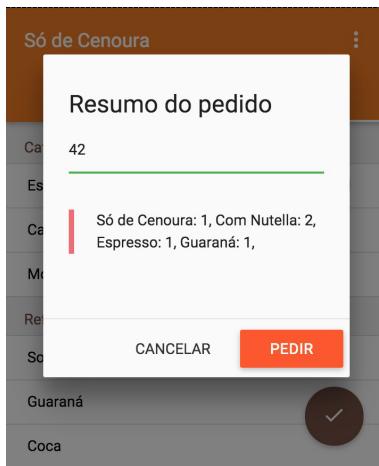
- E um elemento ainda em branco do tipo `blockquote` , que já vai conter o resumo do pedido do usuário:

```
<blockquote id="resumo"></blockquote>
```

Coloque esses três elementos dentro do `modal-content` .

Esse elemento `#resumo` precisa ter uma lista simples dos produtos selecionados e suas quantidades. Vamos implementar isso em JavaScript. Queremos saber todos os itens que possuem `badge` , pois isso significa que eles possuem uma certa quantidade.

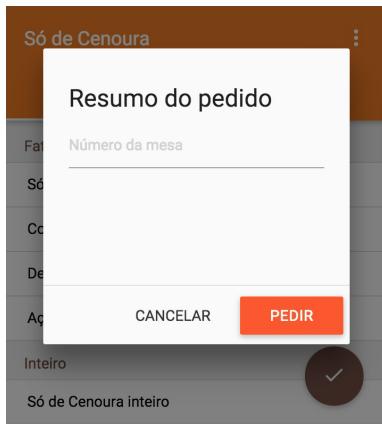
Depois, pegamos os valores dos itens e dos badges e concatenamos em uma String simples de texto. Por fim, esse texto deve ser o valor de `#resumo` . Vale lembrar que tudo isso é quando o nosso botão de ação for acionado (ele tem um ID `#confirmar`).

Em jQuery, fazemos:

```
$('#confirmar').on('click', function() {
    var texto = "";
```

```
    $('.badge').parent().each(function(){
        texto += this.firstChild.textContent + ': ';
        texto += this.lastChild.textContent + ', ';
    });

    $('#resumo').empty().text(texto);
});
```

A barra de ações

O rodapé do modal geralmente possui botões de ação. O comum é ter uma confirmação e um de cancelar. Vamos criar ambos com `<button>` , mas de tipos diferentes. Usaremos um `btn-flat` para o de cancelar, e um `btn` normal para o de confirmar.

Dentro do `modal-footer` , **insira os 2 botões**:

```
<button
   class="btn deep-orange waves-effect waves-light modal-close">
    Pedir
</button>
<button class="btn-flat waves-effect waves-red modal-close">
    Cancelar
</button>
```

Repare que, além das classes de efeitos, colocamos também a `modal-close` . No *Materialize*, isso faz com que o botão feche o modal ao ser acionado.

Teste a App e veja o modal sendo aberto:

Ao clicar no botão **Pedir**, faremos uma chamada Ajax para o back-end e registraremos o pedido. Vamos implementar isso em um capítulo mais à frente. Por enquanto, o botão não faz nada.

9.4 MAIS AJUSTES

Podemos acrescentar pequenos ajustes na aplicação para melhorar seu uso. Por exemplo, limpar um certo produto caso seja pedido errado. Podemos fazer um evento no próprio `badge` que, se acionado, deve ser removido. Com jQuery:

```
$('.collection')
    .on('click', '.badge', function() {
        $(this).remove();
        return false;
    })
```

Outro ponto é se a garçonete quiser apagar o pedido todo. Podemos adicionar um botão **Limpar** que apaga tudo. Em

JavaScript, isso é simples. Se ele tiver uma classe `acao-limpar` , podemos fazer:

```
$('.acao-limpar').on('click', function() {
    $('#numero-mesa').val('');
    $('.badge').remove();
});
```

Mas onde colocar tal ação? Podemos usar o menu de ação secundária que criamos antes - aquele ícone de 3 pontinhos no topo. O submenu em si nada mais é que uma lista de opções dentro de um elemento com classe `dropdown-content` .

Crie esse menu chamando nossa ação *Limpar* logo abaixo do ícone `more_vert` - no HTML, era um elemento `<i>` .

```
<ul id="submenu" class="dropdown-content">
  <li><a class="black-text acao-limpar">Limpar</a></li>
</ul>
```

Para o submenu ser acionado, precisamos **modificar o ícone** e falar que ele dispara esse *dropdown*. Isso é feito apenas com *data attributes* na tag e com uma classe especial `dropdown-btn` . **Altere** a tag do `<i>` para ficar:

```
<i class="material-icons waves-effect waves-light
          waves-circle dropdown-button"
    data-activates="submenu" data-gutter="5"
    data-constrainwidth="false">
        more_vert
</i>
```

Figura 9.4: Dropdown com cara de Material

Podemos aplicar nossa `acao-limpar` ao botão *Cancelar* do modal também. Assim, ao clicar, o pedido é reiniciado.

Código final da App

Deixei o código todo da App disponível neste branch do GitHub:

https://github.com/sergiolopes/garconapp/tree/js.

9.5 TESTANDO NO ANDROID

Lembre-se de que, a qualquer momento, podemos testar no nosso aparelho Android ou no emulador Genymotion apenas rodando:

```
cordova run android
```

9.6 E O IOS?

Uma das garçonetes do **Só de Cenoura** comprou um novo iPhone, e queria usar a nossa *garconapp* no seu novo aparelho. Ainda bem que estamos usando Cordova e HTML! Já estamos a meio caminho andado. No próximo capítulo, veremos como adicionar suporte ao iOS no nosso projeto.

CAPÍTULO 10

PREPARANDO O AMBIENTE PARA IOS

Como vimos no capítulo 5, você vai precisar rodar o OSX, seja em um Mac real, um Hackintosh ou mesmo virtualizado de alguma maneira. Neste capítulo, vamos preparar o ambiente para criar, buildar, emular e testar no iOS usando Mac. Assumo que você já configurou o Cordova com Node.js, como vimos no capítulo 6.

Você vai precisar de uma **Apple ID**, mas provavelmente você já tem uma, visto que ela é essencial para usar um Mac. Se não, crie em: https://appleid.apple.com.

10.1 XCODE

As ferramentas de desenvolvimento da Apple estão todas no guarda-chuva do *Xcode*. Então, precisamos instalá-lo. Baixe o *Xcode* na Mac App Store, ou em:

https://developer.apple.com/xcode/downloads/.

A instalação é padrão do OSX, apenas arrastar para *Applications*. Então abra o **Xcode** pela primeira vez para ele terminar a instalação.

10.2 IOS-SIM E IOS-DEPLOY

O *cordova-ios* usa duas ferramentas Node.js. O **ios-sim** ajuda a

lidar com o emulador, e o **ios-deploy** permite rodar a App em um dispositivo real. Vamos instalar ambas.

No terminal, use o `npm` para instalar globalmente:

```
sudo npm -g install ios-sim ios-deploy
```

10.3 ADICIONANDO CORDOVA-IOS

Para conseguirmos testar nosso projeto no emulador, precisamos adicionar o **cordova-ios** no projeto. Vá ao terminal, entre na pasta do `garconapp` e adicione a plataforma, como fizemos com o Android antes:

```
cd garconapp
cordova platform add ios
```

Feito este passo, conseguimos rodar no emulador usando:

```
cordova emulate ios
```

Se tudo correu bem, ele deve abrir nossa App no emulador padrão:

Para ver quais emuladores há disponíveis, execute:

```
cordova emulate ios --list
```

Você pode escolher um deles na hora de rodar, como por exemplo:

```
cordova emulate ios --target iPad-Air
```

INSTALANDO OUTROS EMULADORES

No Xcode, você pode ir em *Preferences*, aba *Downloads*, e escolher mais versões do iOS para baixar o emulador. É bem útil para testar em versões mais antigas do iOS também.

10.4 TESTANDO EM UM DISPOSITIVO IOS

Para rodar em um dispositivo real, é um pouco mais complicado. O iOS não deixa rodar Apps em modo inseguro, então todas precisam estar assinadas digitalmente em um processo um tanto quanto chato. É importante saber que até no Xcode 6.x era necessário ter uma conta paga da Apple de desenvolvimento para gerar as chaves para assinar a App. No Xcode 7, ele permite gerar chaves de desenvolvimento com uma conta normal, e é o que faremos aqui.

Quando adicionamos a plataforma iOS no Cordova, ele criou um projeto completo no Xcode. Esse projeto fica em `garconapp/platforms/ios/` , e é um arquivo com o nome da App e extensão `.xcodeproj` .

Abra esse arquivo. Ele deve abrir o projeto na interface do Xcode. Clique no nome do projeto no painel da esquerda para mostrar as suas configurações.

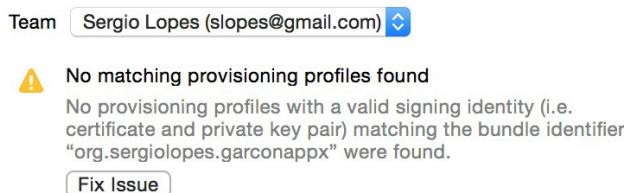
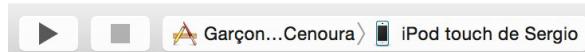



Figura 10.2: Visão do projeto pelo Xcode

Repare no botão de *Play* em cima. Você pode escolher algum dos emuladores e rodar imediatamente. Mas isso não é o que queremos; queremos rodar no dispositivo real. Para isso, precisamos adicionar nossa conta no Xcode e criar um *Provisioning Profile*.

Nesta tela mesmo do projeto, repare que no meio há opções do projeto - como *Bundle identifier*, *Version* etc. Agora nos interessa a configuração **Team**. Se você nunca usou o Xcode antes, ele deve estar vazio. Clique na opção e em **Add an Account**. Siga os passos para fazer login com seu Apple ID.

Adicionada sua conta, ela deve aparecer agora no dropdown de *Team* do projeto. Selecione-o. Ele deve reclamar que falta um *provisioning profile* para essa App. Clique no botão *Fix* que ele gerará os certificados para você.

> É bem importante que você esteja usando um *Bundle identifier* único para seu usuário, se não ele vai acusar um erro. Se precisar mudar, mexa no ID do `config.xml` do Cordova e gere novamente o projeto do Xcode com `cordova prepare ios`.

Se tudo der certo, o seu usuário deve aparecer como *Team*, e não deve ter nenhum *warning* embaixo.

Agora, conecte o dispositivo via USB e desbloqueie a tela. Na aba superior do botão *Play*, deve aparecer seu aparelho, além dos emuladores. Selecione-o e clique em *Play* para executar.

A App é então aberta no aparelho real e você pode testá-la de verdade. Quando acabar, basta apertar o botão *Stop* no Xcode.

Feita toda essa configuração, você consegue também executar agora pelo Cordova na linha de comando. Basta executar:

```
cordova run ios --device
```

10.5 EM OUTROS PROJETOS

Toda vez que criar um projeto iOS e quiser testar no aparelho, você vai precisar seguir esses passos. Abrir o projeto no Xcode, selecionar o *Team* e gerar o *Provisioning profile*.

10.6 E PARA DISTRIBUIR A APP?

Repare que, com todo esse processo que vimos aqui, conseguimos instalar a App em um dispositivo iOS para testes. Você pode instalar em mais de um aparelho repetindo o procedimento, mas não é algo muito confortável.

O iOS não permite a facilidade do Android de mandar uma App para alguém por e-mail e qualquer pessoa poder instalar.

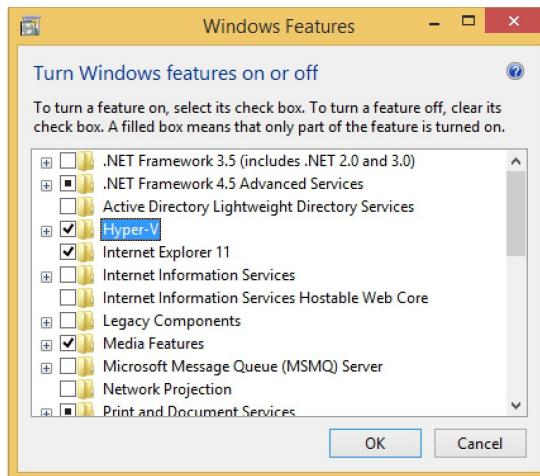
Para distribuir Apps no iOS para todo mundo, precisamos publicá-las na *App Store*. Veremos como fazer isso no fim deste livro.

CAPÍTULO 11

SUPORTE AO WINDOWS PHONE E WINDOWS PLATFORM

O Cordova suporta a criação de aplicativos para a plataforma Windows de várias formas. A partir do Windows 8 e incluindo o Windows 10, o **desenvolvimento é unificado para smartphones, tablets e desktop**. Ou seja, com uma única plataforma, podemos atacar toda a família de dispositivos Microsoft. É o cenário que veremos aqui. No Cordova, é o *Windows Platform*.

11.1 PREPARANDO O AMBIENTE PARA WINDOWS PHONE

Como vimos no capítulo 5, você vai precisar rodar um Windows, seja em uma máquina real ou virtualizado. O detalhe é que, se quiser rodar o emulador, você *precisa* ter o **Windows Pro** ou **Enterprise** na versão **64bits**. E o processador da sua máquina precisa ser um Intel com suporte **Hyper-V**. A maioria dos processadores modernos suporta (i5, i7).

Se você precisar de um Windows, há um trial de 90 dias gratuito da versão *Enterprise* em:

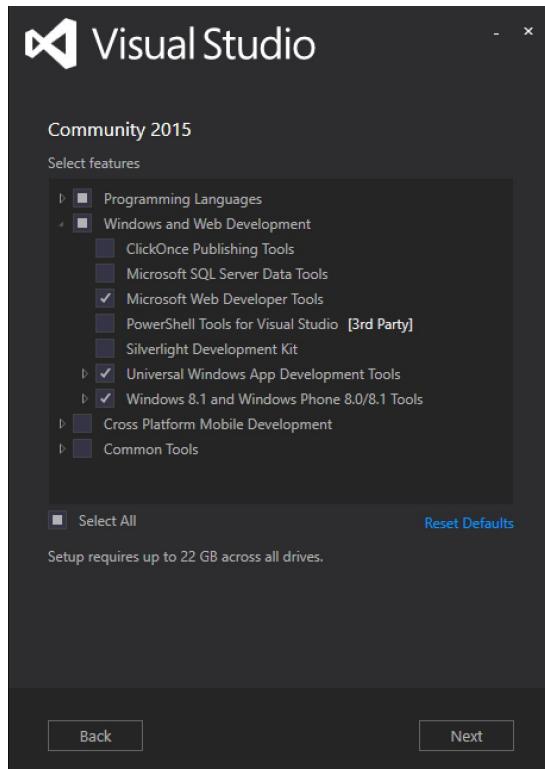
https://www.microsoft.com/en-us/evalcenter/evaluate-windows-10-enterprise.

Habilitando Hyper-V

Para rodar o emulador do Windows Phone, o suporte a **Hyper-V** precisa estar habilitado no seu Windows.

Para isso, vá ao *Painel de Controle* e entre na categoria *Programas*. Lá, selecione a opção **Ativar ou desativar recursos do Windows** (*Turn Windows features on or off*).

Localize o checkbox do **Hyper-V** e marque a opção. É necessário reiniciar após essa configuração.

11.2 VISUAL STUDIO E EMULADORES

Você vai precisar de toda a plataforma **Visual Studio** da Microsoft que já inclui os emuladores necessários.

Você pode baixar uma versão gratuita mais simples chamada de **Visual Studio Community**. É suficiente para nós. Baixe a versão 2015 em:

https://www.visualstudio.com/products/visual-studio-

community-vs.

Rode o instalador e selecione *Custom Installation* para selecionar mais componentes. Estamos interessados nos emuladores mobile. Marque *Universal Windows App Development* e *Windows Phone 8.1*.

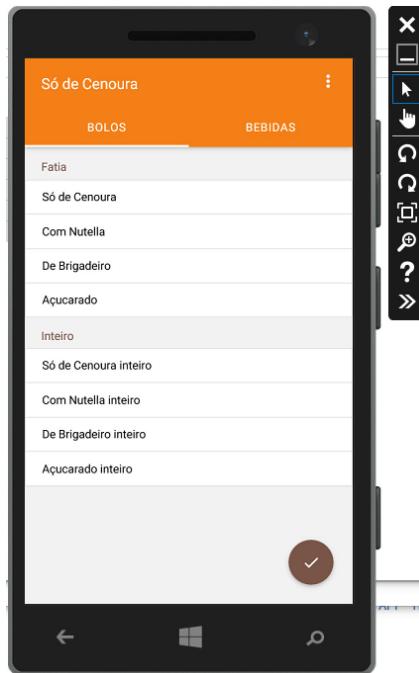
Faça o processo de instalação todo. Ele é bem longo e demorado, e inclui muitos downloads. No final, você deve ter no seu menu *Iniciar* o *Visual Studio* e *Windows Phone Tools*.

Abra o Visual Studio pela primeira vez e faça login com sua conta Microsoft.

11.3 PLATAFORMA WINDOWS NO CORDOVA

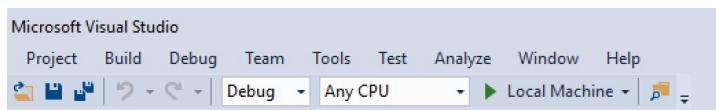
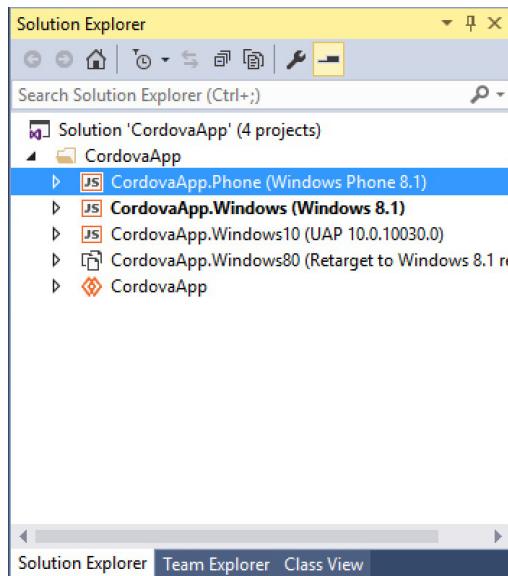
Para adicionar a plataforma portável do Windows 8/10, faça:

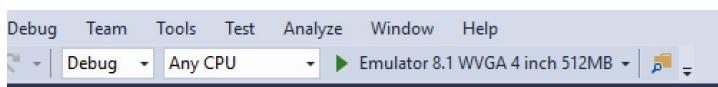
```
cordova platform add windows
```

O interessante é que o desenvolvimento é portável a ponto de você conseguir rodar uma App Cordova direto no Windows Desktop. Basta fazer:

```
cordova run windows
```

A primeira vez que você rodar esse comando, ele pedirá uma série de confirmações. Em uma janela, ele vai pedir para colocar o computador em *Modo Desenvolvedor*. Em outra, pede para obter licenças de desenvolvedor. É só seguir as opções e ir confirmando. No fim, a App abre no Desktop mesmo como uma App normal.

Mas claro que queremos rodar no emulador para simular um aparelho real com Windows Phone. Para fazer isso, execute:

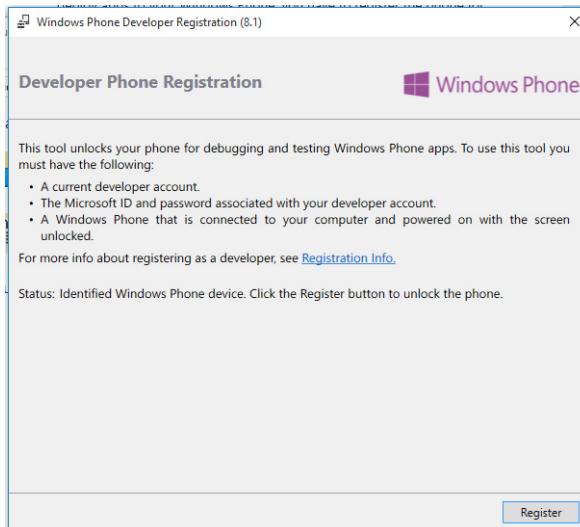
```
cordova run windows -- --phone
```

O emulador do Windows Phone vai abrir e executar a App. A primeira vez demora um pouco para subir, mas você pode deixá-lo aberto e só executar o comando de novo para rodar novamente.

11.4 RODANDO PELO VISUAL STUDIO

Quando adicionamos a plataforma `windows` no Cordova, ele gera um projeto válido do *Visual Studio*. Basta entrar na pasta do projeto e ir em `platforms\windows`. Lá tem um arquivo com nome do projeto do tipo *Visual Studio Solution* (extensão `.sln`). Só clicar e o Visual Studio será aberto.

Se quiser, pode customizar arquivos e configurações específicas do Windows pelo Visual Studio, ou rodar o emulador por aí. No topo da janela, há um botão de *Play* onde você escolhe o destino da execução. Pode selecionar um emulador, a máquina local ou um dispositivo.

Repare que, na direita, há o *Solution Explorer* que lista os projetos da nossa App. E o Cordova gerou um projeto Windows Phone, outro Windows 10 etc. Você pode clicar em cada projeto com o botão direito e escolher *Set as Startup Project* para definir qual é o padrão.

Como estamos preocupados com mobile, selecione o projeto do *Windows Phone*. Repare que o menu de debug muda para mostrar os emuladores disponíveis.

Como vimos, podemos então executar a App tanto pela linha de comando com Cordova quanto com Visual Studio. Rodar pelo Visual Studio tem a vantagem do modo debug e permitir inspecionar os elementos da App, muito útil para desenvolvimento.

11.5 APARELHO WINDOWS PHONE

Para usar um dispositivo com Windows Phone real, são necessárias algumas configurações adicionais, tanto no aparelho quanto no PC. Eu usei um Lumia 520 com Windows 8.1 para o tutorial a seguir.

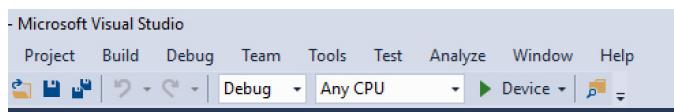
Registrar o aparelho (Windows Phone 8.1)

O aparelho precisa ser registrado na Microsoft para permitir a instalação de Apps de testes. Com uma conta gratuita da Microsoft, conseguimos registrar 1 aparelho. Para registrar mais aparelhos e publicar na loja, você precisará de uma conta paga.

Quando instalamos o Visual Studio e o SDK do Windows Phone, veio junto uma ferramenta chamada **Windows Phone Developer Registration**. Você a encontra no menu *Iniciar*.

Conecte o aparelho via USB, desbloqueie a tela, e abra a ferramenta. Ele deve identificar seu aparelho e dar a opção de *Register*.

Clique em *Register*. Ele vai pedir seu usuário Microsoft e logo registrar o aparelho.

Saiba mais em https://msdn.microsoft.com/en-us/library/windows/apps/dn614128.aspx.

Developer Mode (Windows Mobile 10)

Se seu celular já roda Windows 10, não é preciso mais registrá-lo. Vá nas *Configurações* e em *Atualização & Segurança*. Lá existe uma opção *Para Desenvolvedores*, na qual você pode habilitar o *Modo Desenvolvedor*.

Para saber mais, acesse https://msdn.microsoft.com/en-us/library/windows/apps/dn706236.aspx.

Rodando pelo Visual Studio

Ao abrir o projeto no Visual Studio, como fizemos antes, o botão de *Play* lá em cima mostra a opção *Device*. Clique para rodar.

Rodando via Cordova

Além de rodar pelo Visual Studio, você pode rodar pela linha de comando com Cordova também. Basta executar:

```
cordova run windows --device -- --phone
```

11.6 E PARA DISTRIBUIR A APP?

O que vimos aqui é útil para desenvolvedores testarem suas Apps. Mas se você quiser distribuir a App para outras pessoas, não é

tão prático. O mais prático, claro, é publicar na *Windows Store*.

Um processo intermediário é gerar um instalável e distribuí-lo nos dispositivos. Não é tão simples quanto o Android, mas pelo menos existe uma possibilidade (diferente do iOS).

Primeiro, gere o build final da App, um arquivo `.appx`, com:

```
cordova build windows --release
```

Ele vai gerar um arquivo `appx` na pasta `platforms\windows\AppPackages\GarconappTest`. Podemos instalá-lo no aparelho com outra ferramenta que foi instalada junto com o SDK, a **Windows Phone Application Deployment**. Antes, esteja certo de ter destravado o dispositivo como fizemos antes. Assim, abra essa ferramenta, selecione o arquivo `.appx` e clique em **Deploy**.

Capítulo 12

CONFIGURAÇÕES DO PROJETO

Quando você testar nossa App nos emuladores ou nos dispositivos, verá que não temos um ícone bonito, nem uma splash screen e nem o nome da App. No capítulo 3, discutimos o arquivo `config.xml` e como configurar várias dessas coisas.

O *Cordova* também usa um `config.xml` , mas ele é ligeiramente diferente da versão do *PhoneGap*. Vamos ver como fazer no Cordova.

Primeiro, a localização do arquivo que muda. No *Cordova*, por padrão, ele fica na **raiz do projeto**. Se preferir, pode colocar dentro de `www/` como no *PhoneGap*, que ele também suporta.

Quando geramos o projeto, o Cordova gerou um arquivo padrão cheio de pré-configurações. Vamos aprender a configurar nosso projeto agora. Então **apague o `config.xml` gerado**.

12.1 ESTRUTURA DO CONFIG.XML

A própria declaração do arquivo muda um pouco por conter um namespace do *Cordova*:

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="org.sergiolopes.garconapp"
        version="0.0.1"
        xmlns="http://www.w3.org/ns/widgets"
```

```
        xmlns:cdv="http://cordova.apache.org/ns/1.0">
</widget>
```

Dois atributos aqui são importantes:

- `id` : contém um identificador único para nosso projeto. É usual colocar o seu domínio ao contrário no início para evitar bater com o id de outra pessoa. Esse id é usado nas lojas das plataformas para publicação. *Lembre de usar um id único para a sua App.*
- `version` : um número de versão seguindo o padrão *SEMVER*. A ideia é incrementá-lo conforme for lançando as versões da App.

12.2 METADADOS DO PROJETO

As informações básicas são iguais ao que vimos antes do PhoneGap. *Autor*, *Nome da App* e *Descrição* podem ser feitos com:

```
<name>Garçonete Só de Cenoura</name>

<description>
    App de uso das garçonetes do
    restaurante Só de Cenoura.
</description>

<author href="http://sergiolopes.org"
        email="qualquer@sergiolopes.org">
    Sérgio Lopes
</author>
```

O `<name>` vai aparecer no menu do usuário e nas lojas. As outras infos são importantes para a publicação nas lojas. Teste a App novamente e já teremos o nome atualizado.

Há várias outras configurações opcionais. Por exemplo, se o HTML principal da sua App **não** for o `index.html` , você pode usar o `<content>` para indicar:

```
<content src="minha-app.html" />
```

12.3 PREFERÊNCIAS

Assim como no PhoneGap, podemos usar várias tags `<preference>` para habilitar e desligar certos comportamentos. Por exemplo:

```
<preference name="Orientation" value="portrait" />
<preference name="Fullscreen" value="true" />
```

Conseguimos travar a orientação da App em `portrait` ou `landscape`. Já a opção *Fullscreen* esconde a barra de status no topo e deixa em tela cheia (padrão `false`).

Podemos também mudar a cor de fundo padrão da App. Usamos `BackgroundColor` que recebe um valor hexadecimal em RGBA. Cuidado, ele recebe mais 2 dígitos que o normal na Web para indicar o canal Alpha. Use `FF` para cores opacas.

```
<preference name="BackgroundColor" value="0xF2F2F2FF" />
```

Outras preferências controlam mais pequenos detalhes da App. A `DisallowOverscroll`, por exemplo, impede no iOS e no Android que o scroll estoure os limites da App - por padrão, você vai reparar que ele deixa a App ser scrollada para fora e depois volta.

```
<preference name="DisallowOverscroll" value="true" />
```

Veja outras preferências globais na documentação:

https://cordova.apache.org/docs/en/dev/config_ref/index.html

12.4 PREFERÊNCIAS POR PLATAFORMA

É possível mudar as preferências também por plataforma. Por exemplo, para deixar a App em tela cheia apenas no Android, faríamos:

```
<platform name="android">
    <preference name="Fullscreen" value="true" />
</platform>
```

Ou para desabilitar o *overscroll* apenas no iOS:

```
<platform name="ios">
    <preference name="DisallowOverscroll" value="true" />
</platform>
```

12.5 PREFERÊNCIAS ESPECÍFICAS DE PLATAFORMA

Algumas preferências são específicas de plataforma e permitem configurar detalhes bem particulares de cada uma. No iOS, por exemplo, temos a `TopActivityIndicator` , que permite configurar a cor (`white` , `gray` , `whiteLarge`) daquele *spinner* que aparece na barra de status quando há atividade de rede.

Muitas outras preferências podem ser mudadas no iOS. Veja mais em:

http://cordova.apache.org/docs/en/dev/guide/platforms/ios/config.html

No Android, é mesma coisa; há mais um monte de preferências específicas. Por exemplo, podemos controlar quanto tempo a *splash screen* deve aparecer com `SplashScreenDelay` (o padrão é 3 segundos).

Você pode consultar as muitas outras preferências configuráveis no Android em:

http://cordova.apache.org/docs/en/dev/guide/platforms/android/config.html

12.6 ÍCONE E SPLASH SCREEN

O ícone de uma App é muito importante. Ele é usado na home do usuário, na loja da plataforma e em vários outros lugares. **Toda App precisa de um ícone.** Se não indicamos um, o Cordova usa um padrão dele.

Outro conceito importante em Apps é a tela de carregamento, chamada de *splash screen*. É uma imagem de tela cheia que aparece enquanto a App está carregando, para o usuário não ficar com a tela em branco. **Toda App precisa de uma splash screen.** E aqui também o Cordova coloca uma padrão caso não especifiquemos.

Mas como colocar nosso próprio ícone e *splash screen*? Parece bem simples. Só criar os arquivos PNG e apontar no `config.xml` :

```
<icon src="resources/icon.png" />
<splash src="resources/splash.png" />
```

Note que usei a pasta `resources` . Podia ser qualquer uma, mas esse é um nome bastante usual. Perceba também que esse caminho é relativo à raiz do projeto, não à pasta `www` .

O que parece um simples XML de duas na linhas na prática **é muito mais complicado**. Como existem muitos aparelhos diferentes, com telas e resoluções diferentes, precisamos de arquivos específicos para cada tipo. E são *muitas* possibilidades.

Enquanto escrevo o livro, só para o iOS, precisamos de **16 tamanhos diferentes** do ícone. O menor em 40 x 40 pixels até o maior em 180 x 180 pixels - e mais 14 tamanhos intermediários. No Android, são muitos mais, e é assim também para as outras plataformas.

Na *splash screen*, é mesma coisa. Há dezenas de tamanhos diferentes e, pior, tudo em versão retrato e paisagem (já que o arquivo não é quadrado).

Se você tiver paciência para exportar todas essas imagens, ainda

vai precisar configurar o XML. Só os primeiros 5 ícones do iOS ficariam:

```
<icon src="resources/ios/icon-60@3x.png" width="180" height="180"
/>
<icon src="resources/ios/icon-60.png" width="60" height="60" />
<icon src="resources/ios/icon-60@2x.png" width="120" height="120"
/>
<icon src="resources/ios/icon-76.png" width="76" height="76" />
<icon src="resources/ios/icon-76@2x.png" width="152" height="152"
/>
```

Está curioso? 180 é para o iPhone 6 Plus, 60 é para o iPhone normal e 120 o de retina. Aí 76 para o iPad normal e 152 para o iPad retina. E isso só para o iOS 7 em diante. Se quiser suportar iOS 6 e outros, você precisará de mais um monte. Ou seja, é insano gerenciar os ícones de uma App Cordova na mão.

12.7 AUTOMATIZANDO GERAÇÃO DOS ÍCONES E SPLASH SCREENS

Queremos uma ferramenta que, a partir de um arquivo principal, gere todos os tamanhos de ícones e splash screens necessários. E, importantíssimo, gere a configuração no `config.xml`.

Existem vários projetos no GitHub com esse propósito. Eu vou usar o **Ionic** para isso. Vamos falar mais do *Ionic* no livro mais à frente. Ele é um framework com muitas ferramentas e serviços úteis para Apps Cordova. Por hora, vamos usar um pequeno serviço que ele oferece: a geração dos ícones e splash screens.

Instalação do Ionic

O *Ionic* é baseado no node.js e distribuído via `npm`, assim como Cordova. Precisamos instalá-lo primeiro. Abra o terminal e faça:

```
npm -g install ionic
```

Lembre-se de que no Mac e no Linux, você precisará do `sudo` na frente do comando.

Preparando os arquivos

O Ionic espera que você crie dois arquivos no projeto:

```
resources/icon.png
resources/splash.png
```

Eles serão a base para rodar a geração dos demais tamanhos. Por isso, devem ser arquivos bem grandes, pois serão redimensionados.

O `icon.png` deve ser quadrado e seu desenho deve ocupar toda a área útil. O tamanho recomendado é de 1024 x 1024px ou maior. Não adicione efeito algum, pois cada plataforma aplica seu efeito. As bordas redondas no iOS, por exemplo, são feitas pela plataforma; o ícone deve ser quadrado.

Já o `splash.png` também deve ser quadrado e bem grande. Recomendo 4096px, mas o desenho é diferente. A ideia é usar alguma ilustração centralizada e longe das bordas. No fim, o arquivo será cortado para encaixar em cada plataforma.

Eu preparei os 2 arquivos base para o *Só de Cenoura*, e você pode baixá-los nesta pasta do GitHub:

https://github.com/sergiolopes/garconapp/tree/config/resources

.

Executando o Ionic Resources

Uma vez criados os dois arquivos na pasta `resources` - e com os nomes `icon.png` e `splash.png` -, basta rodar o Ionic. Bem

simples.

Abra um terminal, entre na pasta do projeto e rode:

```
ionic resources
```

Ele gerará todos os ícones necessários para cada plataforma que estivermos usando. E ainda vai atualizar o `config.xml` apropriadamente.

Se você quiser baixar o projeto já com os ícones gerados e tudo configurado, deixei-os em uma branch separada no GitHub:

https://github.com/sergiolopes/garconapp/tree/iconesgen.

SUPORTE AO WINDOWS PLATFORM

Enquanto escrevo o livro no começo de 2016, o Ionic Resources ainda não suporta a plataforma Windows, só iOS e Android. Então, gerei os arquivos necessários e coloquei o XML correspondente manualmente. Você pode baixar no link anterior do GitHub.

Mais Ionic Resources

Existem mais algumas opções e configurações possíveis. Consulte a documentação:

http://ionicframework.com/docs/cli/icon-splashscreen.html.

Uma que pode ser interessante, dependendo do projeto, é gerar ícones diferentes para cada plataforma. Basta criar uma subpasta com o nome da plataforma e o arquivo `icon.png` (ou `splash.png`).

```
resources/android/icon.png
resources/ios/icon.png
```

Ao rodar `ionic resources` novamente, ele vai gerar ícones de todos os tamanhos, mas usando uma base diferente para cada plataforma.

12.8 ENGINES

Todo projeto Cordova possui diversas pastas importantes. Nós trabalhamos bastante com a `www` e com o `config.xml` , mas há outras como a `platforms` , que possui os códigos e projetos nativos de cada plataforma que adicionamos.

Essa pasta `platforms` inclusive é bem grande e é gerada automaticamente pelo Cordova quando adicionamos alguma plataforma. Podemos até mexer nos códigos nativos, mas evitamos isso. Muita gente não gosta de mandar esse monte de código gerado para o repositório do projeto, pois fica com sujeira demais.

Uma opção, então, é não commitar essa pasta - adicionar no `.gitignore` , por exemplo. Mas e quando uma pessoa nova precisar do projeto, como ela sabe quais plataformas estamos usando e suportando?

Para resolver isso, você pode adicionar no `config.xml` :

```
<engine name="android" />
```

A `engine` é uma plataforma em uso. Quando alguém baixar o projeto, a primeira vez que rodar `cordova prepare` , a plataforma indicada será baixada e gerada automaticamente.

Podemos colocar várias plataformas e até indicar qual versão deve ser usada exatamente:

```
<engine name="android" spec="5.0.0" />
<engine name="ios" spec="4.0.1" />
```

Se você já instalou as plataformas como fizemos nos capítulos anteriores, usando no terminal `cordova platform add android` , ele não gerou esses XMLs. Porém, é fácil colocá-los agora, basta rodar:

```
cordova platform save
```

E nas próximas vezes que adicionar uma plataforma nova, você também pode fazer:

```
cordova platform add android --save
```

CAPÍTULO 13

PLUGINS NO CORDOVA

O dono do *Só de Cenoura* teve uma ideia genial para facilitar ainda mais o trabalho das garçonetes que vão usar a App para anotar pedidos. Em vez de digitar o número da mesa a cada pedido, ele colou pequenas etiquetas com *qrcodes* nas mesas indicando seu número.

Com isso, basta a garçonete apontar a câmera para o *qrcode*, e o número da mesa será preenchido automaticamente. Quer dizer, é isso que ele quer, então é nosso trabalho agora fazer **a App escanear qrcodes**.

13.1 MAIS PODER COM CORDOVA

Até o momento, se for pensar bem, nossa App é apenas um HTML, CSS e JS normal. Podia muito bem ser uma *WebApp*. A única coisa que o Cordova nos deu até agora foi o empacotamento nativo e a possibilidade de instalar como uma App normal. Mas o Cordova permite muito mais do que isso.

A grande vantagem do Cordova é permitir expor **recursos nativos** do aparelho e da plataforma para nosso código JavaScript. Isso quer dizer acessar funcionalidades que não estão disponíveis para páginas Web comuns, mas apenas para Apps nativas.

Isso é feito com a adição de **plugins**. Há centenas de plugins diferentes no mercado, alguns oficiais do Cordova, outros oficiais da

Adobe e muitos outros da comunidade. Com eles, conseguimos acessar todos os sensores do aparelho - GPS, giroscópio, acelerômetro etc. -, obter informações do sistema - nível de bateria, sinal de rede -, integrar com dados nativos - como os contatos do usuário - e acessar hardware - câmera, cartão de memória, bluetooth etc. E muito mais, na verdade. E, claro, existe um **plugin para leitura de qrcodes**.

13.2 PHONEGAP BARCODE SCANNER

A Adobe tem um plugin para leitura de códigos diversos, incluindo *qrcodes*, códigos de barra e outros vários. Ele é opensource e você encontra código e documentação aqui:

https://github.com/phonegap/phonegap-plugin-barcodescanner.

Como adicionar esse plugin no Cordova? Pela linha de comando, é bem simples:

```
cordova plugin add phonegap-plugin-barcodescanner
```

Ele vai baixar e instalar o plugin em todas as plataformas que estivermos usando. Você pode até olhar na pasta `plugins` do projeto e o verá instalado lá.

Também podemos indicar esse plugin no `config.xml` e evitar commitar a pasta `plugins` no repositório. Basta usar a tag `plugins` que recebe qual versão queremos ou asteriscos para indicar a última:

```
<plugin name="phonegap-plugin-barcodescanner" spec="*" />
```

Desta forma, a próxima vez que rodarmos o Cordova, ele vai baixar e configurar o plugin nas plataformas existentes no projeto.

Outra opção é instalar e configurar o XML ao mesmo tempo,

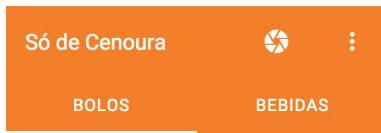
com:

```
cordova plugin add phonegap-plugin-barcodescanner --save
```

13.3 LENDO CÓDIGOS DE BARRAS

O plugin oferece uma API JavaScript bastante simples para ler códigos de barra. Basta chamar `cordova.plugins.barcodeScanner.scan` e passar um callback para receber o resultado.

```
cordova.plugins.barcodeScanner.scan(function(resultado) {
    alert(resultado.text);
});
```

As APIs do Cordova ficam penduradas nesse objeto `cordova` global, onde são carregados os plugins que vamos instalando. Esse plugin de qrcode em particular tem uma função `scan` bem simples.

Mas claro que não vamos só chamar o código assim e pronto. Precisamos colocar um botão na interface e tratar o retorno, inserindo o valor no formulário.

13.4 INTEGRANDO NA APP

Vamos primeiro adicionar um botão na nossa App para disparar o leitor de qrcodes. Crie um novo ícone no topo da App para a câmera.

```
<i class="material-icons waves-effect waves-light waves-circle sca
n-qrcode">camera</i>
```

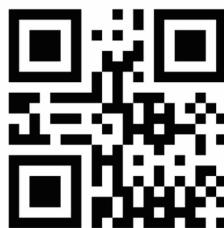
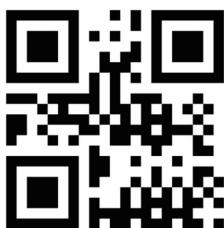
Coloque dentro do `div` do cabeçalho, logo antes do outro ícone `<i>` que já temos lá. Repare que usamos as classes do *Materialize* para o ícone ficar bonito, e colocamos uma classe nossa chamada `scan-qrcode`.

Abra o arquivo `app.js` que tínhamos criado e vamos implementar a funcionalidade com jQuery. Quando o usuário acionar o botão do ícone, disparamos o código do `barcode Scanner`. Após escanear, pegamos o valor e inserimos no campo do formulário (`#numero-mesa`), além de mostrar um `Toast` simples de confirmação.

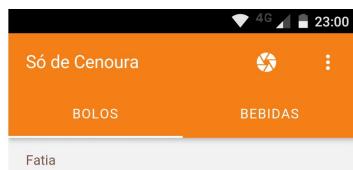
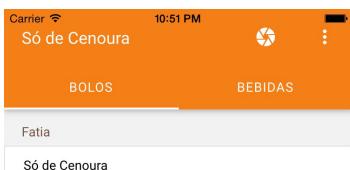
```
$('.scan-qrcode').on('click', function(){
    cordova.plugins.barcodeScanner.scan(
       function (resultado) {
           if (resultado.text) {
               Materialize.toast('Mesa ' + resultado.text, 2000);
               $('#numero-mesa').val(resultado.text);
           }
       },
       function (error) {
           Materialize.toast('Erro: ' + error, 3000, 'red-text');
       }
    );
});
```

Se você testar agora, vai reparar que não funciona. Ele reclama que não encontra o objeto `cordova`, justo aquele que deveria nos dar acesso ao plugin. O problema é que faltou **importar o arquivo** `cordova.js`, que é quem faz a ponte com o código nativo.

Basta importar o script na página logo antes dos nossos scripts:

```
<script src="cordova.js"></script>
```

Note, porém, que não temos esse arquivo no projeto. É isso mesmo. Esse arquivo é **gerado pelo Cordova** automaticamente na hora do build da App.

Observe também que, ao usar um plugin nativo como esse,

precisamos testar em um dispositivo real, para ver o comportamento funcionando. Se tentarmos abrir o `index.html` direto no navegador, o botão não vai funcionar, já que não há suporte ao plugin, que é nativo.

Então rode no seu aparelho e use esses *qrcodes* para testar:

13.5 UMA BOA STATUS BAR

Uma característica comum a boas Apps nativas é integrar bem ao visual da plataforma em que está rodando. E um ponto importante é a **barra de status**, aquela barra do topo com horário, notificações etc.

Se você abriu nossa App no iOS e no Android, deve ter visto isso:

No iOS, a *status bar* fica por cima da App, quase atrapalhando o conteúdo da App. No Android, a barra de status é preta e não atrapalha a App. Mas nenhum desses comportamentos é o usual da plataforma.

Veja, por exemplo, o Google Inbox aberto no iOS e no Android:

No iOS, o comum é ter uma status bar de cor sólida na mesma cor do topo da aplicação e, claro, não por cima da App. No Android, o comum é ter uma barra de status em uma cor secundária, um tom mais escuro da cor do topo da App.

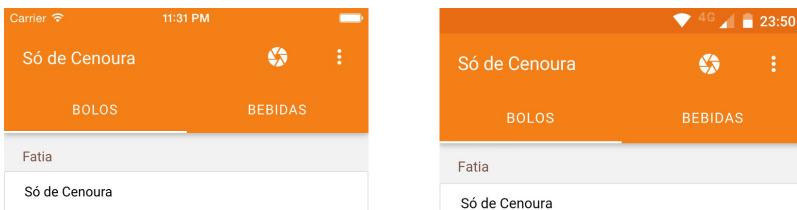
Queremos configurar essas características na nossa App. Para isso, precisamos de um plugin que lida com a status bar.

13.6 CONFIGURANDO A STATUS BAR

Instale o plugin pelo terminal. É um plugin oficial do Cordova:

```
cordova plugin add cordova-plugin-statusbar --save
```

Esse plugin permite muitas customizações, como cores e comportamento da status bar. Permite até escrever código para esconder e mostrar programaticamente. Nosso interesse agora é configurar o seu visual, e isso o plugin permite com **preferências específicas no** `config.xml` .

Por exemplo, para indicar no iOS que a barra não deve ficar por cima da App, usamos a preferência `StatusBarOverlaysWebView` com valor `false` (o padrão é `true`). Para mudar a cor, usamos `StatusBarBackgroundColor` com o valor de uma cor em hexadecimal como na Web.

A questão é lembrar de que cada plataforma tem sua característica. No iOS, deve ser a mesma cor do topo da App e, no Android, um tom mais escuro. Podemos controlar isso com a tag `<platform>` que vimos no capítulo anterior.

A configuração final fica:

```
<platform name="android">
    <preference name="StatusBarBackgroundColor" value="#E86C13" />
</platform>

<platform name="ios">
    <preference name="StatusBarOverlaysWebView" value="false" />
    <preference name="StatusBarBackgroundColor" value="#F57F17" />
</platform>
```

Teste novamente e teremos agora:

Projeto final do capítulo

Se preferir o projeto deste capítulo com o qrcode e status bar está nesta branch no GitHub:

https://github.com/sergiolopes/garconapp/tree/plugins.

CAPÍTULO 14

SERVIÇOS REMOTOS

Falta um último detalhe para nossa App poder ser usada nos restaurantes: efetivamente finalizar o pedido. Fizemos a listagem de produtos e a seleção dos itens do pedido, mas precisamos efetivá-lo, enviar para a cozinha preparar.

O *Só de Cenoura* já possui uma solução back-end na cozinha. É uma aplicação Web que mostra em um monitor grande quais são os pedidos em preparo. E essa aplicação tem um **serviço REST** onde podemos **registrar novos pedidos**.

Vamos usar esse serviço na nossa App para registrar no sistema da cozinha os pedidos feitos nas mesas.

14.1 A API DA COZINHAPP

A **cozinhapp** é uma aplicação Web em Node.js e Meteor bastante simples. Possui dois propósitos: mostrar os pedidos em aberto para os cozinheiros e expor uma API REST para registro de novos pedidos. Vamos observar essas coisas primeiro antes de integrar na App.

O **painel de pedidos abertos** está rodando em:

http://cozinhapp.sergiolopes.org

Abra no seu navegador. É uma lista simples de pedidos - possivelmente vazia, se não houver pedidos novos agora.

Para registrar os novos pedidos, existe um *endpoint REST* em http://cozinhapp.sergiolopes.org/novo-pedido. Ele exige a passagem de dois argumentos: `mesa` com o número da mesa que fez o pedido, e `pedido` com o texto do pedido. Bem simples.

Abra outra aba no navegador e teste chamando na mão mesmo:

http://cozinhapp.sergiolopes.org/novo-pedido?mesa=17&pedido=Bolo+de+Cenoura

Se olhar na lista, seu pedido vai aparecer agora. Você pode inserir outros mais como teste para ver o sistema funcionando. Clique em *Finalizar* para remover o pedido da lista.

> **Como funciona a aplicação Web**
>
> Essa parte do back-end foi escrita em Node.js com Meteor. Nosso foco, claro, é na parte mobile e a integração com a nossa App Cordova. Então, vamos usar o serviço pronto. Se quiser, o código está disponível em
>
> https://github.com/sergiolopes/cozinhapp.
>
> Um detalhe importante é que temos apenas um back-end para todos os leitores do livro. Isso quer dizer que seus pedidos podem se misturar a pedidos de outras pessoas. Se preferir, rode sua própria instalação isolada da cozinhapp.

14.2 AJAX NA APP

O que precisamos, então, é acionar essa API via Ajax. Podemos usar jQuery para isso. É um código sem segredo, um simples Ajax mesmo. Veja um exemplo:

```
$.ajax({
    url: 'http://cozinhapp.sergiolopes.org/novo-pedido',
    data: {
        mesa: $('#numero-mesa').val(),
        pedido: $('#resumo').text()
    }
});
```

Estou usando o `$.ajax` do jQuery que recebe a `url` e os parâmetros na opção `data`. Repare que já estou passando os valores reais com base nos dados da nossa App.

Mas é claro que esse código precisa ser disparado apenas quando o botão *Pedir* do modal for acionado. Vamos adicionar uma classe `acao-finalizar` a esse botão, que deve ficar mais ou menos assim:

```
<button class="btn deep-orange waves-effect waves-light modal-clos
e acao-finalizar">Pedir</button>
```

E, com jQuery, pegamos o clique do botão:

```
$('.acao-finalizar').on('click', function() {
    // código do Ajax vai aqui depois
});
```

Além disso, podemos colocar *callbacks* de **sucesso** e **erro**. O serviço traz uma mensagem de resposta que vamos mostrar no componente **toast** do *Materialize*. No callback de sucesso, também vamos limpar os campos para permitir um novo pedido.

O código completo para colocarmos no `app.js` é:

```
$('.acao-finalizar').on('click', function() {
    $.ajax({
        url: 'http://cozinhapp.sergiolopes.org/novo-pedido',
        data: {
            mesa: $('#numero-mesa').val(),
            pedido: $('#resumo').text()
        },
        error: function(erro) {
           Materialize.toast(erro.responseText, 3000, 'red-text');
        },
        success: function(dados) {
```

```
            Materialize.toast(dados, 2000);

            $('#numero-mesa').val('');
            $('.badge').remove();
        }
    });
});
```

Apesar de correto, se você testar esse código agora, ele não vai funcionar. Temos algumas **restrições de segurança** que precisamos seguir.

14.3 CORS E SAME ORIGIN POLICY

Os navegadores Web têm uma restrição de segurança famosa, chamada *Same Origin Policy*. Ela faz com que certos requests, como os feitos via Ajax, só sejam permitidos dentro do mesmo domínio.

Ou seja, se nosso serviço está em `cozinhapp.sergiolopes.org` , por padrão, o Ajax só vai funcionar em outras páginas que também estejam em `cozinhapp.sergiolopes.org` . O que não é o caso da nossa App no Cordova.

Nosso HTML no Cordova roda localmente no celular do usuário. Na prática, é como abrir um arquivo local direto no browser. Ele usa URLs `file://` , o que não bate com a origem do serviço. Então, o request é bloqueado.

Há diversas formas de resolver essa limitação. A solução definitiva e recomendada é usar **CORS** (*Cross-Origin Resource Sharing*). A implementação é simples e envolve apenas o servidor - o browser entende automaticamente.

Basta, no servidor, colocar um cabeçalho novo da resposta, `Access-Control-Allow-Origin` , indicando quais outros domínios (origens) estão permitidos para acessar aquele serviço. Por

exemplo:

```
Access-Control-Allow-Origin: http://sodecenoura.com.br
```

Isso abre o serviço para um outro domínio hipotético: http://sodecenoura.com.br.

O problema com Cordova é **o que colocar nesse cabeçalho**. Não existe uma origem `http` , um domínio nas Apps Cordova. Lembre-se de que elas rodam em `file://` . Para resolver, precisamos **liberar o CORS para todos** com:

```
Access-Control-Allow-Origin: *
```

Isso libera o acesso a todas as origens, o que inclui nossa App Cordova. Uma ressalva é que isso não significa que seu back-end está menos seguro. Para segurança, usamos autenticação (com OAuth por exemplo). O CORS só relaxa o nível de acesso para o *cliente*, o que nesse caso não é um problema.

Nos seus serviços, portanto, **você vai precisar acrescentar o cabeçalho do CORS no back-end**. A nossa aplicação Node já está configurada com esse cabeçalho para facilitar. Você pode até inspecionar no navegador e vê-lo em ação.

Mas se o CORS já estava habilitado no nosso back-end, por que ainda não funcionou? O Cordova bloqueia por padrão todas as requisições externas. Precisamos liberar explicitamente.

14.4 PLUGIN CORDOVA WHITELIST

Existe um plugin chamado `cordova-plugin-whitelist` , que controla o acesso a URLs externas na nossa App Cordova. Ele vem, inclusive, instalado por padrão quando criamos o projeto. Ou podemos instalá-lo com:

```
cordova plugin add cordova-plugin-whitelist --save
```

Com ele, podemos listar no `config.xml` todas as origens que queremos liberar acesso. Para liberar nossa API back-end, colocamos:

```
<access origin="http://cozinhapp.sergiolopes.org" />
```

A tag `access` recebe uma origem. Você pode usar asteriscos para vários subdomínios - por exemplo, `http://*.sergiolopes.org`. Você pode também listar subpastas apenas, ou até liberar para todas as URLs do mundo colocando apenas `*`, o que não é uma boa ideia, a menos que realmente precise.

Coloque então essa tag `access` no `config.xml`, liberando nosso back-end. Depois, teste a App no aparelho novamente. Tudo deve funcionar agora.

Nota sobre iOS 9 e App Transport Security Settings

A partir do iOS 9, todas as Apps, tanto nativas quanto híbridas, só podem fazer chamadas a serviços seguros `https://` por padrão. Isso quer dizer que um Ajax para `http://`, como fizemos, não funcionaria.

Uma das coisas que o `cordova-ios` faz é abrir exceções para cada domínio necessário, relaxando as configurações de segurança caso a caso, com base nas configurações do *whitelist*. Também é possível deixar aberto para todas as URLs, mas não é recomendado.

Você pode encontrar essas configurações geradas no arquivo `platforms/ios/Garconete/Resources/Garconete-Info.plist`. Lá, procure pela chave `NSAppTransportSecurity`. Você pode customizar esse arquivo também.

Se preferir, existe uma versão alternativa do serviço usado neste capítulo que roda em https: **https://cozinhapp.meteor.com**.

14.5 INTENTS E NAVEGAÇÕES

Nossa App usa apenas o whitelist de chamadas Ajax. Entretanto, há outras configurações possíveis com esse plugin. Vejamos, só para conhecimento.

Como o Cordova roda em um navegador, nada impede que cliquemos em um link que leve para outra página. Por padrão, só podemos navegar em páginas locais de `file://` . Porém, às vezes, vamos querer navegar para outras URLs. Liberamos isso com:

```
<allow-navigation href="http://sergiolopes.org/*" />
```

Também podemos listar quais URLs devem ser tratadas pelo sistema operacional, e não pelo navegador. É o que chamamos de **Intents**. Certas URLs especiais podem abrir Apps nativas do sistema, e precisamos liberar quais queremos. Por exemplo, podemos liberar links em URLs do tipo `tel:` para abrir o discador nativo:

```
<allow-intent href="tel:*" />
```

Veja mais em https://github.com/apache/cordova-plugin-whitelist.

CAPÍTULO 15

ARQUITETURA DO CORDOVA

Já fizemos bastante coisa com Cordova, e aprendemos vários de seus conceitos na prática ao longo do desenvolvimento do projeto. Mas é bom agora pausarmos um pouco as novas funcionalidades e elaborar um pouco mais como o Cordova realmente funciona. Entender sua arquitetura, suas vantagens e limitações.

15.1 O QUE SÃO AS WEBVIEWS

Todas as plataformas móveis oferecem, além do navegador normal de internet, uma espécie de navegador nativo sem interface, que pode ser usado pelas Apps. São as **WebViews**. Pense nela como um navegador capaz de executar todo HTML, CSS e JavaScript normalmente, mas sem a interface de um navegador, e sem botões, barra de endereço, menus etc. **Só o renderizador**.

O objetivo é que Apps nativas da plataforma possam usar esse componente de WebView para exibir conteúdo HTML em algum ponto da App. O Facebook, por exemplo, usa WebView pra renderizar os links clicados em sua App sem abrir o browser completo.

Pense na WebView como um motor de renderização de HTML plugável. No iOS, ele é baseado no WebKit como o Safari. No Android, é baseado no Chromium a partir da versão 4.4, e no

WebKit em versões anteriores. No Windows 10, é baseado no EdgeHTML como o Microsoft Edge - e usa o Trident do IE no Windows Phone 8. Vamos ver mais detalhes sobre as WebViews no capítulo 19.

15.2 UMA APP CORDOVA

Muitas Apps usam WebViews para exibir pequenos conteúdos HTML em seu fluxo. Porém, o Cordova inovou ao renderizar a **App inteira dentro de uma WebView**.

Uma App Cordova é uma App nativa do sistema operacional que tem apenas uma função: chamar a WebView e carregar o arquivo `index.html` dentro dela. Toda a interface do usuário, toda lógica da App será escrita em HTML, CSS e JS e executada dentro da WebView, assim como uma WebApp.

A ideia é tão simples que, se você conhecer um pouquinho de Android e Java, conseguiria fazer essa casca simples do Cordova em pouquíssimas linhas de código:

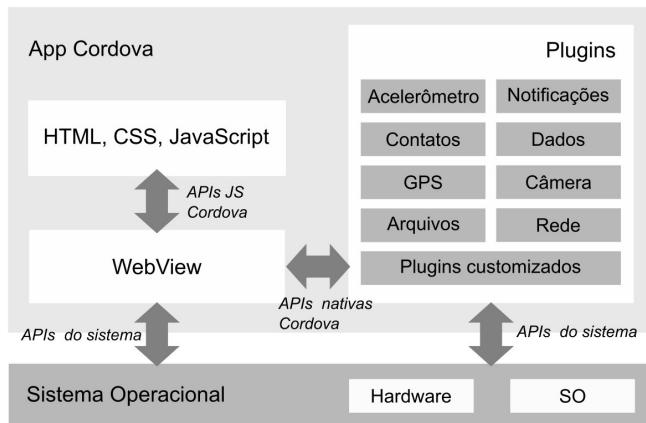
```
<WebView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview" />

WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.loadUrl("www/index.html");
```

Nas outras plataformas, o código é diferente, mas a ideia é muito parecida e simples.

O que o Cordova faz é escrever esses códigos nativos de cada plataforma e empacotar junto com seu código HTML da pasta `www` . **O resultado é uma App nativa que invoca a WebView e carrega seu `index.html` dentro dela**.

Inclusive, se você abrir a pasta `platforms` dentro do projeto

Cordova, verá o código nativo completo cuspido pelo Cordova em cada plataforma.

15.3 WEBVIEW COM PODERES

Uma WebView normal tem a mesma capacidade de um navegador Web comum. Isso quer dizer que não pode acessar recursos nativos da plataforma. O Cordova dá um passo além. Ele expõe uma API JavaScript que pode ser chamada da WebView e chama recursos nativos.

Hoje cada recurso nativo especial é encapsulado em um plugin diferente. Então, se você quiser, por exemplo, acessar os contatos do aparelho, instale o plugin *cordova-plugin-contacts*. Esse plugin tem todo o código nativo necessário para acessar os contatos nativos em Java para Android, Objective-C para iOS etc. E expõe para a WebView o objeto `navigator.contacts` , onde você pode manipular os contatos de forma portável. Por exemplo:

```
var contato = navigator.contacts.create({"displayName": "Olga"});
```

A vantagem é que **toda a complexidade nativa está encapsulada no plugin**, e nós escrevemos JavaScript normal na página sem problemas.

Esse gráfico ilustra como uma App Cordova funciona:

A App Cordova é um pacote que inclui nosso HTML, CSS e JS, junto com os plugins que estivermos usando e a chamada da WebView nativa. Nosso código chama APIs em JavaScript na WebView que pode disparar chamadas nos plugins. Estes, por sua vez, se comunicam com o sistema operacional de forma nativa.

Repare que, além dos plugins prontos, podemos, claro, criar nossos **próprios plugins**. Não é uma tarefa tão simples quanto apenas usar um pronto, mas é possível. Precisamos de um certo conhecimento das plataformas nativas para isso.

15.4 ESTRUTURA DE UM PROJETO CORDOVA

Já navegamos bastante nas pastas de um projeto Cordova, porém, é interessante entender para que serve cada pasta e arquivo exatamente.

- `www` - é onde fica todo seu código HTML, CSS e JavaScript, além das imagens e outros recursos estáticos.
- `config.xml` - o arquivo de configurações do Cordova. Fica na raiz do projeto.

- `platforms` - pasta gerada onde ficam os projetos nativos de cada plataforma em uso. Geralmente, não mexemos nessa pasta, ou mexemos muito pouco. Um exemplo:
    - `plataforms/android` - possui todo o código gerado para Android com uma cópia da nossa `www` e um projeto completo do Android Studio.
- `plugins` - pasta onde ficam os plugins instalados no Cordova. Não é costume commitar no projeto.
- `resources` - pasta comum onde colocamos ícones e splash screens para todas as plataformas.
- `hooks` - pode incluir scripts para serem executados durante os procedimentos do Cordova. Vem vazia e não é necessária.

15.5 CORDOVA NÃO É WEB

Falamos um pouco disso no começo do livro, mas vale reforçar: o Cordova não é Web. Usamos HTML, CSS e JavaScript para escrever a interface, mas não estamos na Web. Não temos um servidor HTTP, não temos URLs públicas, nada disso.

Nossos arquivos empacotados dentro da App Cordova são acessados pela WebView como `file://` . Você pode até testar isso observando o `location.href` .

15.6 CARA DE APP, JEITO DE APP

O principal desafio de uma App Cordova é **parecer uma App**. Como usamos HTML, é muito fácil escorregar e fazer algo muito parecido com uma página Web comum.

Essencialmente, precisamos de um design com cara de App, uma interação com jeito de App, acesso a recursos nativos e boa performance.

Design appy

Quando usamos o framework *Materialize*, nosso objetivo foi trazer o design do Material Design do Android para nosso HTML. O visual final é bem parecido com uma App Android normal.

Em geral, é isso o que precisamos buscar. **Um visual que transmita a ideia de App** e se integre ao tipo de visual que o usuário está acostumado nas plataformas móveis.

Isso não significa sempre usar um visual com cara nativa, como o Materialize. Inclusive, como a App é multiplataforma, pode ser mais interessante um visual não vinculado a uma plataforma específica.

Vários frameworks no mercado têm componentes visuais prontos em HTML e CSS para você criar Apps lindas. Alguns dos mais famosos hoje são:

- Ionic - http://ionicframework.com/
- Onsen UI - http://onsen.io/
- ChocolareChip UI - http://chocolatechip-ui.com/
- Framework7 - http://www.idangero.us/framework7/

Até o próprio Bootstrap pode ser interessante, dependendo do cenário. O importante é focar no visual com cara de App.

Single Page Applications

Só o visual, porém, não é suficiente. **A interação com a App deve parecer uma App.** E o principal aspecto é não atualizar a tela a cada navegação. Na Web, estamos acostumados a clicar em um link ou botão e isso abrir uma nova página do zero.

Em Apps, não. Acionamos um botão e a próxima tela é carregada dinamicamente, muitas vezes até com transições

animadas.

Em termos práticos, isso quer dizer que não é uma boa ideia criar as telas da sua aplicação como arquivos HTML separados, navegando entre si. O que temos é um HTML inicial que carrega as telas sob demanda usando Ajax e com animações CSS bonitas entre as navegações.

Isso é o que chamamos de **Single Page Applications**. Um único HTML que carrega a App toda e cuida de toda navegação.

Você pode fazer uma SPA simples com um pouco de JavaScript e jQuery, mas, na prática, há frameworks melhores para isso no mercado. Hoje, os mais famosos são o Angular e o React, mas temos ainda Ember, Backbone e outros.

Melhor ainda quando você une um bom framework SPA com uma biblioteca de componentes visuais bonitos. E esse é o caso de vários frameworks que falamos antes, em especial o Ionic.

O **Ionic** usa o Angular por baixo e coloca seus componentes ricos em cima. É hoje o framework mais famoso para esse tipo de SPA mobile como as que queremos fazer no Cordova. Mas há vários outros. Vamos ver um pouco de Ionic no próximo capítulo.

Se você reparar bem nas duas Apps que fizemos até agora, ambas, apesar de simples, eram *single page applications*. Um único HTML lidava com todo o conteúdo e todas as telas, sem navegação externa.

A *garconapp*, por exemplo, permitia navegar entre as abas, abrir o menu, fazer o pedido, tudo sem atualizar a tela toda. Como era uma App simples, não tínhamos telas secundárias carregadas via Ajax, mas a ideia é a mesma. Em uma App mais complexa, use algum framework JavaScript de SPA robusto.

Recursos nativos

Além de bom visual e interação com jeito de App, queremos **uma boa integração com os recursos da plataforma**. Isso vai desde um bom ícone e uma boa splashscreen até acesso a recursos de hardware avançados.

Queremos usar as notificações do sistema. Queremos acessar câmera, acelerômetro, giroscópio e outros recursos. Queremos uma status bar bonita integrada ao sistema. Queremos acessar os contatos do aparelho e manipular arquivos no cartão de memória. Enfim, tudo o que os diversos **plugins do Cordova** oferecem. Precisamos apenas usá-los e pensar em boas experiências para o usuário.

15.7 BOA PERFORMANCE É ESSENCIAL

Outro ponto bastante importante é performance. Há muito a se falar sobre isso, que mereceria um livro só para esse assunto. Mas vamos discutir alguns cenários importantes.

Toda App precisa de boa performance para agradar o usuário. Entrtanto, a questão aqui é que o **Cordova coloca alguns obstáculos a mais** , nos fazendo começar já em desvantagem. E se nossa App não parece tão rápida quanto as demais Apps do aparelho do usuário, ele vai reclamar.

A principal questão é que a *WebView* é um interpretador de HTML, CSS e JavaScript. E isso é inerentemente mais lento do que qualquer coisa feita nativamente na plataforma. Isso é um fato. O que precisamos é cercar o máximo possível os gargalos para ter uma performance aceitável em cada caso. A maioria das Apps não precisa de ultra performance, precisa mais é *parecer* rápida.

Resposta ao toque

Um ponto de atenção é a resposta ao toque. Os primeiros navegadores móveis vinham com um atraso proposital no toque dos botões da ordem de 300ms. Não parece muito, mas eles causam uma demora perceptível entre o usuário acionar um botão e a ação executar. Os navegadores e *WebViews* modernos já retiraram essa restrição, mas antigos ainda têm.

A solução mais comum é usar o **Fastclick** (https://github.com/ftlabs/fastclick). Os frameworks completos como Ionic já cuidam disso para você também, o que pode ser mais um bom motivo para adotá-los.

Animações

Interfaces ricas costumam ter belas animações. O ideal é fazê-las em **CSS usando aceleração 3D**. Isso significa transições que usem apenas `transform` e `opacity` . Pesquise também sobre o `will-change` .

Outra opção é frameworks JavaScript de animações performáticas, como o *Greensock* ou *Velocity.js*. Fuja do jQuery para animações. E, novamente, frameworks completos como Ionic já trazem animações comuns em componentes e transições de tela.

Processamento pesado

A maior limitação do JavaScript hoje é que ele roda em uma única thread. Isso quer dizer que todo o seu código JS, todos os eventos do usuário e muitas coisas da interface do navegador/WebView são executados em uma única thread de maneira intercalada.

Basta um código lento para travar tudo. Você já deve ter

aberto uma página com scroll lento, ou clicado em um botão que teima em não responder, ou visto uma animação que não roda de forma fluída. Geralmente, esses problemas são causados pela thread única do JavaScript estar ocupada executando coisas demais. O pessoal chama isso de **jank** (http://jankfree.org/).

O recado é evitar código que demorem demais para executar. Mais que uns poucos milissegundo, já é problema. Quebre os códigos em pedaços menores que você pode agendar para executar aos poucos com `setTimeout` ou `requestAnimationFrame` .

Se precisar processar coisas pesadas, use *Web Workers*, que são a solução JavaScript para executar coisas fora da thread principal. Tem suas limitações, mas é o que temos disponível.

Offline-first

Uma App empacotada com Cordova já funciona offline por padrão. Todo HTML, CSS e JS está instalado junto com a App e pode ser acessado sem rede. Mas boa parte das Apps acessa serviços externos via Ajax, por exemplo, o Facebook que precisa pegar as informações atualizadas do Feed de Notícias.

O ideal é construir uma experiência offline-first. Isso significa salvar localmente os dados baixados da rede e usá-los na App caso não tenhamos rede. Melhor ainda: usar os dados locais sempre e, caso tenha rede, buscar dados atualizados de forma transparente em background. É assim que o Facebook e outros funcionam. E isso dá uma ótima sensação de performance para o usuário.

Vamos implementar uma ideia assim no capítulo 18.

15.8 RECURSOS INTERESSANTES DA API DO CORDOVA

O Cordova expõe alguns recursos interessantes para a WebView que podemos acessar via JavaScript no nosso código. São recursos que permitem interagir melhor com o aparelho e com o próprio ambiente da WebView.

Eventos do Cordova

O próprio Cordova dispara alguns **eventos adicionais** no JavaScript em momentos importantes para uma aplicação híbrida. O mais famoso deles é o `deviceready`, que é chamado quando as APIs do Cordova estão prontas para uso.

Às vezes, queremos chamar um recurso do Cordova logo no início da App, mas como ele tem um componente nativo, ele pode não estar disponível ainda. O `deviceready` nos indica que tudo do Cordova foi inicializado, e que podemos chamar as APIs e plugins sem problemas. Exemplo:

```
document.addEventListener("deviceready", function(){
    // acessa alguma coisa do navigator.contacts por exemplo
});
```

Outros eventos importantes são do **ciclo de vida da App**. O evento `pause` é disparado quando o sistema operacional suspende nossa App, em geral, porque o usuário alternou para outra App. É um bom momento para suspender qualquer coisa pesada que estejamos processando, ou mesmo parar animações. O evento `resume` é disparado quando a App volta à ativa, e aí podemos acionar novamente os recursos que foram pausados.

Podemos também escutar eventos de botões nativos do sistema operacional, por exemplo, `volumedownbutton` e `volumeupbutton`. Nas plataformas que têm botão de voltar, como Android e Windows Phone, podemos escutar o evento `backbutton` também.

Há alguns outros eventos, que você pode consultar em:

https://cordova.apache.org/docs/en/latest/cordova/events/events.html.

Merges

Um recurso simples, mas bem poderoso, são os *merge folders*, que permitem customizar **códigos específicos para cada plataforma**. 99% do nosso código deve ser portável, mas às vezes queremos fazer algo específico, como esconder o botão de back no Android, já que ele já possui o nativo; mudar a fonte usada no Windows Phone, ou qualquer outra coisa.

Imagine que temos um `detalhes.css` que queremos que seja diferente em certas plataformas. No HTML, usamos:

```
<link rel="stylesheet" href="css/detalhes.css">
```

Podemos criar um arquivo `www/css/detalhes.css` padrão a ser usado em todas as plataformas. Mas aí criar versões específicas em uma nova pasta `merges/` na raiz do projeto - ao lado da `www` - com subpastas com as plataformas interessadas. Por exemplo, no Android, seria: `merges/android/css/detalhes.css` , e assim para as demais plataformas.

No momento do build da App, o Cordova verifica os arquivos da pasta `merges` daquela plataforma, e substitui os arquivos correspondentes na `www` somente naquela plataforma. É possível até ter arquivos específicos novos para apenas uma plataforma sem nem aparecer no `www` , como um ícone de voltar apenas no iOS em `merges/ios/img/back.png` .

CAPÍTULO 16

INTRODUÇÃO AO IONIC

A próxima fase do *Só de Cenoura* é lançar uma App simples para delivery. Os clientes querem comprar os deliciosos bolos e receber em casa. A nova App deve suportar o máximo de plataformas possíveis, permitir visualizar o cardápio de entrega e fazer um pedido.

Por ser uma App multiplataforma, focada nos clientes, vamos evitar usar o *Material Design*, que tem um apelo maior apenas no Android. Para facilitar, queremos um framework de componentes mobile com visual agnóstico de plataforma. E, de preferência, que já se integre ao Cordova e ao estilo de criação de Apps híbridas. Vamos usar o **Ionic Framework** (http://ionicframework.com/).

16.1 SOBRE O IONIC

O Ionic nasceu como um **framework de componentes visuais com foco em mobile** construído em cima do Angular. Isso quer dizer então que ele segue todo o modelo de *Single Page Application* do Angular, ideal para Apps híbridas.

Os componentes foram criados com foco em um visual independente de plataforma e usabilidade otimizada para mobile e touch screens. Existem dezenas de componentes prontos e você pode consultá-los aqui:

http://ionicframework.com/docs/components/.

Você pode usar o Ionic apenas pegando o CSS e o JS deles e importando em sua página. Mas eles também criaram uma **CLI**, interface em linha de comando, similar ao Cordova, que facilita várias tarefas.

É possível criar um projeto novo com Ionic e já ter todos os recursos importados, e testar usando o Cordova em devices reais e emuladores. **Um projeto Ionic é um projeto Cordova com superpoderes.** Então, tudo o que fizemos no Cordova até agora é possível ser feito também no projeto Ionic.

O *Ionic Framework* é o ponto principal, mas ele cresceu para algo conhecido como **Ionic Platform**, que adiciona alguns serviços adicionais além do framework base em si.

Já usamos um desses serviços no capítulo 12, o **Ionic Resources**. Por meio da linha de comando do Ionic, geramos os ícones e splash screens do projeto usando esse serviço - que gera tudo remotamente nos servidores deles.

Há outros serviços, alguns pagos:

- **Ionic View**: permite uploadar seu projeto para o servidor do Ionic e visualizá-lo em qualquer dispositivo. Muito útil para testes (http://view.ionic.io/).
- **Ionic Creator**: interface Web para criar Apps Ionic com arrastar e soltar (https://creator.ionic.io).
- **Ionic Market**: distribuição de plugins e temas para Apps Ionic (http://market.ionic.io/).
- Além disso, novos serviços de Push Notifications, Analytics, empacotamento de Apps e mais (http://www.ionic.io/).

Como o foco do livro não é no Ionic nem no Angular em si, vamos ver apenas algumas de suas funcionalidades. A ideia é usar o

conceito de *single page application* em uma App híbrida com Cordova.

16.2 NOVO PROJETO

Vamos iniciar uma nova App para os clientes fazerem seus pedidos de casa, a *pedidapp*.

Se você não instalou o Ionic antes, faça-o agora. É bem simples. No terminal:

```
npm install -g ionic
```

No terminal, usaremos o comando `ionic start` com alguns argumentos. Em `--appname`, indicamos o nome da App; em `--id`, o identificador único do pacote (use um próprio), depois a pasta onde gerar e, por último, o template base a ser usado (*blank* indica um projeto mínimo).

```
ionic start --appname "Só de Cenoura em Casa" --id org.sergiolopes
.pedidapp pedidapp blank
```

Agora entre na pasta `pedidapp`, e execute o preview do projeto:

```
cd pedidapp
ionic serve
```

O navegador abrirá e veremos um preview simples da App. Basta irmos modificando a App agora e o navegador será **atualizado automaticamente**. No fim, veremos como executar no dispositivo.

16.3 ARQUITETURA DA APP

Abra a pasta `pedidapp` no seu editor e vá se familiarizando com o código. Ele é um projeto Cordova com mais coisas do Ionic. De nosso interesse, agora, é a pasta `www`. Lá você vai ver um HTML

simples e um arquivo `app.js` que inicializa o Angular e o Ionic.

Observe o `index.html` e verá alguns componentes do Ionic dentro do `<body>`. Você pode até testar mudar o valor do `<h1>` e ver o projeto atualizado no navegador.

Vamos querer usar outros componentes do Ionic e montar nossa App completa. A princípio, ela terá 3 telas principais, com diversas funcionalidades:

- **Home**: lista de todos os produtos que o usuário pode pedir.
- **Detalhe Bolo**: mostra os detalhes de um bolo específico clicado na Home, com foto e descrição, além de opção de pedir.
- **Pedido**: selecionado o produto, essa tela pede os dados de entrega e registra o pedido.

São 3 telas em que queremos navegar no melhor estilo *Single Page Application*. Queremos animações para as transições de tela, queremos o botão de voltar funcionando, tudo com Ajax sem atualizar a tela toda.

O Ionic vai fazer tudo isso para nós.

Cada tela será um arquivo HTML separado, um *template*. Usamos um arquivo JavaScript para configurar qual template deve ser carregado em qual tela; isso é chamado de *rotas* no Angular. Cada tela pode executar certos códigos através de *controllers*. Além disso, podemos ter *services* para conter lógica comum, como obter os dados da App.

O parágrafo anterior foi um resumo do resumo do resumo de como funciona uma App Angular. Não vamos nos aprofundar tanto na teoria. Vamos aprender na prática.

Comece criando alguns arquivos em branco necessários para nossa App:

- `js/routes.js`
- `js/controllers.js`
- `js/services.js`
- `templates/home.html`
- `templates/detalhe.html`
- `templates/pedido.html`

Agora edite o arquivo `index.html` para incluir os 3 novos arquivos JS que criamos logo antes de fechar o `</head>` :

```
<script src="js/routes.js"></script>
<script src="js/controllers.js"></script>
<script src="js/services.js"></script>
```

Se preferir, baixe o ZIP do projeto com esses arquivos já criados (em branco), em:

https://github.com/sergiolopes/pedidapp/tree/6d85.

16.4 LISTA DE BOLOS

Nossa *Home* vai listar todos os bolos disponíveis para compra em casa. Para criarmos essa tela, precisamos de 3 coisas:

1. A tela em si, seus componentes, precisam ser criados no `templates/home.html` .
2. Criamos um controller em `js/controller.js` para carregar os dados e qualquer lógica necessária para essa tela. Ele se chamará `HomeController` .
3. Amarramos o arquivo do template com o controller com uma URL nas rotas, por meio do arquivo `routes.js` .

Vamos por partes.

A primeira rota

Abra o arquivo `js/routes.js` e adicione a rota `/` para representar a `home`. Indique o arquivo do *template* e o *Controller*. O código será:

```
angular.module('starter')
.config(function($stateProvider, $urlRouterProvider) {

  $urlRouterProvider.otherwise('/home');

  $stateProvider

  .state('home', {
    url: '/home',
    templateUrl: 'templates/home.html',
    controller: 'HomeController'
  })

});
```

Repare que a rota aponta uma `templateUrl` com o nome do HTML a ser carregado. Mas carregado onde? Precisamos indicar no `index.html` onde esses templates vão ser exibidos. Fazemos isso com a tag `<ion-nav-view>`.

Além disso, conforme o usuário navega, queremos exibir o título da página no topo, além de um mecanismo de navegação (*back*). No Ionic, são as tags `<ion-nav-bar>` e `<ion-nav-back-button>`.

Abra o `index.html` e substitua todo o miolo do `<body>` por:

```
<ion-nav-bar>
  <ion-nav-back-button></ion-nav-back-button>
</ion-nav-bar>

<ion-nav-view></ion-nav-view>
```

Aproveite e crie um controller vazio também, se não a rota não funciona. Para isso, vá ao arquivo `js/controllers.js` e coloque:

```
angular.module('starter')
.controller('HomeController', function($scope) {
```

```
});
```

O template

Podemos colocar qualquer HTML no `home.html` e ele será exibido na tela toda assim que a home carregar. Teste se quiser, colocando um `<h1>` simples por exemplo dentro do `home.html` .

O que queremos é uma lista de bolos com um título bonito. Vamos usar componentes do Ionic para isso. Na documentação deles, você encontra a lista completa de componentes:

http://ionicframework.com/docs/.

Uma tela com uma lista simples de bolos usando os componentes do Ionic fica assim:

```
<ion-view>
    <ion-nav-title>Nossos Bolos</ion-nav-title>
    <ion-content>
        <ion-list>
            <ion-item>Só de Cenoura</ion-item>
            <ion-item>Com Nutella</ion-item>
            <ion-item>Com Brigadeiro</ion-item>
            <ion-item>Açucarado</ion-item>
        </ion-list>
    </ion-content>
</ion-view>
```

O `<ion-view>` representa uma nova tela. As telas possuem um conteúdo (`<ion-content>`) que, no caso, é uma lista (`<ion-list>`) de itens (`<ion-item>`).

O `<ion-nav-title>` é um título a ser usado em cada tela. Ele será usado pelo `<ion-nav-bar>` que colocamos antes no `index.html` . Se não fez ainda, rode essa App e verá uma lista simples com 4 opções de bolos além do título *Bolos*.

Um toque de design: podemos colocar a barra de título na cor

laranja do *Só de Cenoura*. Abra o arquivo `css/style.css` e adicione:

```
.bar {
    background: #F77F00;
    color: white;
}
.bar .header-item {
    color: white;
}
```

Você pode ver o código e baixar o ZIP no GitHub em:

https://github.com/sergiolopes/pedidapp/tree/f6e0.

16.5 DADOS DINÂMICOS

Em vez de colocar os nomes dos bolos direto no HTML, o jeito Angular é deixar os dados isolados em um objeto modelo e apenas referenciá-lo no HTML. Com isso, é mais fácil gerenciar os dados - eles podem até vir de uma API Ajax, por exemplo -, e é mais fácil gerenciar o HTML, que fica independente dos dados.

Isso é o **MVC**. A view (HTML) consome o model (os dados) e quem faz a ligação é o controller.

Dentro do `HomeController`, criamos uma variável com um array de objetos que representam os nossos produtos. A princípio,

quero colocar os nomes e preços de cada bolo.

```
angular.module('starter')
.controller('HomeController', function($scope) {

    $scope.bolos = [
        { nome: "Só de Cenoura", preco: 18 },
        { nome: "Com Nutella", preco: 29 },
        { nome: "De Brigadeiro", preco: 24 },
        { nome: "Açucarado", preco: 19 }
      ];

});
```

Expor uma propriedade na variável `$scope` do Angular significa deixar aqueles dados disponíveis na view. Justo o que queremos.

No HTML, podemos usar o `ng-repeat` do Angular para percorrer essa lista de bolos, e exibir os dados usando a notação `{{bolo.preco}}`. O código da lista acessando os dados dinamicamente do controller fica assim:

```
<ion-list>
    <ion-item ng-repeat="bolo in bolos">
        <h2>{{bolo.nome}}</h2>
        <p>apenas R$ {{bolo.preco}}</p>
    </ion-item>
</ion-list>
```

Repare que tiramos os `<ion-item>` fixos e trocamos por apenas um com `ng-repeat`. Ele percorre o array de `bolos`, e monta cada item mostrando o `nome` e o `preco` de cada bolo.

Teste novamente.

16.6 SERVIÇO REST

A mudança para dados dinâmicos foi boa, mas queremos o próximo passo. **Esses dados devem vir do servidor do restaurante.**

Temos já uma API simples que devolve um JSON com essas informações em:

http://cozinhapp.sergiolopes.org/produtos.

Vamos chamar essa URL usando alguns recursos do Angular.

Service e $http

O Angular possui uma diretiva bastante útil para chamadas Ajax, o `$http` . Chamar nossa URL é bem simples:

```
$http.get('http://cozinhapp.sergiolopes.org/produtos')
```

Ela devolve uma *promise* que será executada quando o serviço retornar. Podemos, por exemplo, recuperar os dados da resposta com:

```
$http.get('http://cozinhapp.sergiolopes.org/produtos')
     .then(function(response){
        return response.data;
     })
```

Queremos encapsular essa chamada para que outros lugares da nossa App possam ter acesso a esses dados. É para isso que usamos os *Services* do Angular.

No arquivo `js/services.js` , crie um `ProdutosService` que possui apenas um método `lista` que encapsula justamente esse código que vimos antes:

```
angular.module('starter')
.service('ProdutosService', function($http, $q) {

    var url = 'http://cozinhapp.sergiolopes.org/produtos';

    return {
        lista: function(){
            return $http.get(url).then(function(response){
                return response.data;
            });
        }
```

```
    };

});
```

Agora, queremos trocar no `HomeController` para usar esse novo serviço. No `js/controller.js` , substitua por:

```
angular.module('starter')
.controller('HomeController', function($scope, ProdutosService) {
    ProdutosService.lista().then(function(dados){
        $scope.bolos = dados;
    });
});
```

Repare como recebemos o novo `ProdutosService` como argumento e aí chamamos seu método `lista()` . Ele devolve uma *promise* que resolve com os dados do serviço Ajax que colocamos no escopo para consumo da view.

Teste novamente a App no navegador e veja o serviço sendo chamado.

16.7 TESTE NO DEVICE OU EMULADOR

Apesar de usarmos o Ionic direto no browser, ele já vem com toda estrutura do Cordova preparada. Podemos adicionar uma nova plataforma com:

```
ionic platform add android
```

E depois executar no aparelho Android com:

```
ionic run android
```

Mesma coisa para iOS e as outras plataformas.

Repare também que o projeto possui o `config.xml` padrão do Cordova. Você pode modificá-lo à vontade. Por exemplo, para liberar acesso apenas a URL do nosso serviço, com a tag `<access>` :

```
<access origin="http://cozinhapp.sergiolopes.org" />
```

E, claro, podemos customizar todo o resto. Colocar o nome correto em `<name>`, inserir o `<author>` etc.

O código final desse capítulo está no GitHub para você baixar:

https://github.com/sergiolopes/pedidapp/tree/3bbf.

16.8 TESTANDO COM IONIC VIEW

Se você não tiver o ambiente do Android ou iOS configurado para gerar os pacotes, pode usar o serviço do **Ionic View** para testar suas Apps no dispositivo.

É um serviço semelhante ao *Phonegap Developer App* que vimos no capítulo 4. Mas, diferentemente do Phonegap, seu código fica na nuvem do Ionic, o que permite testar em qualquer aparelho e até mandar para amigos.

A ideia é instalar uma App do Ionic pela própria loja de aplicativos. Aí subimos nosso código no servidor do Ionic e ele executa esse código na App do *Ionic View*. É bastante útil.

O passo a passo é:

1. Crie uma conta em https://apps.ionic.io.
2. Faça upload da sua App Ionic pelo terminal com `ionic upload`.
3. Instale a App do Ionic View no seu dispositivo (http://view.ionic.io/).
4. No device, logue no Ionic View, selecione a App e faça `Download files` e depois `View app`.

É possível também compartilhar a App com outros usuários para eles pré-visualizarem com o *Ionic View*. Basta rodar `ionic share amigo@email.com`.

CAPÍTULO 17

SINGLE PAGE APP COM IONIC

Vamos continuar a App do capítulo anterior com Ionic. Por enquanto, temos apenas a *Home* feita. Vamos criar mais telas, com novas rotas, novos templates e navegar entre elas.

A ideia é explorar o conceito de *Single Page App* onde temos apenas um HTML base (`index.html`) e um framework JavaScript (Angular) que carrega as novas telas dinamicamente via Ajax.

17.1 DETALHES DO PRODUTO

Queremos que, ao clicar em um bolo na *Home*, seja exibida uma tela secundária com os detalhes daquele bolo: uma descrição, uma foto bonita e um botão para fazer o pedido. Para criar uma tela nova, temos sempre o misto: um template HTML, um *controller* e uma rota ligando os dois.

A rota

Comecemos pela rota. Ao navegar para `/bolo` , queremos exibir o detalhe do bolo clicado. Mas como saber qual bolo foi clicado? Precisamos passar um parâmetro ou usar uma rota mais específica. Vamos por esse caminho.

Criaremos uma rota dinâmica no formato `/bolo/0` , `/bolo/1`

etc. A ideia é passar o ID do bolo no fim da URL. Para criar essa rota dinâmica, vá ao arquivo `js/routes.js` e **acrescente** uma nova rota logo embaixo da rota da home:

```
.state('detalhe', {
  url: '/bolo/:boloId',
  templateUrl: 'templates/detalhe.html',
  controller: 'DetalheController'
})
```

Repare na novidade: a `url` possui um parâmetro dinâmico `:boloId`. É uma variável que vai receber aquele número que passaremos na URL.

O controller

Próximo passo é criar o `DetalheController` no `js/controllers.js`. Ele terá o papel de pegar o `id` do bolo e buscar o objeto correspondente nos dados dos bolos.

Com Angular, para pegar o parâmetro definido na rota, fazemos `$stateParams.boloId`. Como nosso array está ordenado, basta acessar a posição correspondente dentro dele. No fim, o novo `DetalheController` completo será:

```
angular.module('starter')
.controller('DetalheController',
  function($scope, ProdutosService, $stateParams) {

    ProdutosService.lista().then(function(dados){
        $scope.bolo = dados[$stateParams.boloId];
    });
});
```

O controller expõe o objeto `bolo` para view. Usaremos seus dados para desenhar a tela bonita de detalhes do bolo.

A view

O HTML da view fica no arquivo `templates/detalhe.html`.

Vamos fazer um design simples a princípio. Fique à vontade para elaborar mais.

Ele mostrará a descrição do bolo, um botão para compra e a foto. Nosso serviço JSON, aliás, já havia devolvido a foto de cada bolo embedada no meio dos dados. Usamos o mecanismo de *data URIs* para devolver as fotos de uma só vez.

Outra abordagem seria indicar apenas a URL de cada foto, e então baixar a foto nessa nova URL. Optamos pela *data URI* no serviço para simplificar.

O arquivo `detalhe.html` ficará:

```
<ion-view>
    <ion-nav-title>{{bolo.nome}}</ion-nav-title>
    <ion-content>
        <img ng-src="{{bolo.foto}}" class="detalhe-foto">
        <p class="detalhe-descricao">{{bolo.descricao}}</p>

        <a class="button button-positive button-full" href="#">
            Pedir esse bolo agora
        </a>
    </ion-content>
</ion-view>
```

Colocamos algumas classes nossas nesse HTML para podermos fazer um layout simples no CSS. Adicione ao arquivo `css/style.css` :

```
.detalhe-foto {
    max-width: 100%;
}

.detalhe-descricao {
    padding: 1em;
    font-size: 120%;
    line-height: 1.5;
}
```

Você pode testar agora navegando manualmente para `#/bolo/0` .

Link na home

Nossa home ainda não linka para a página de detalhes. Faremos isso agora, colocando o atributo `href` na `<ion-item>`.

Aproveitemos também para melhorar o design na home acrescentando uma miniatura da foto. Basta adicionar a classe `item-thumbnail-left` e acrescentar a `<img>`.

Edite o arquivo `home.html` e troque o `<ion-item>` anterior por essa versão com o link e a imagem:

```
<ion-item ng-repeat="bolo in bolos" href="#/bolo/{{bolo.id}}"
          class="item-thumbnail-left">
    <img ng-src="{{bolo.foto}}" style="object-fit: cover">
    <h2>{{bolo.nome}}</h2>
    <p>apenas R$ {{bolo.preco}} {{bolo.quantidade}}</p>
</ion-item>
```

Teste abrir a Home agora e clicar em um dos bolos. A navegação é disparada internamente no Angular, o Ionic faz uma bela animação. Além disso, até um botão *voltar* é acrescentado no topo

da App - importante no iOS que não tem *back* nativo.

O código até aqui está no GitHub:

https://github.com/sergiolopes/pedidapp/tree/da24.

17.2 FINALIZANDO O PEDIDO

A última tela que faremos é a de finalização do pedido, onde o cliente coloca seus dados de entrega e faz o pedido efetivamente.

Nesta primeira versão da App, faremos um pedido simples de apenas um bolo. Não vamos implementar um carrinho de compras para suportar vários produtos, por enquanto. A ideia é simples: na tela de detalhe do bolo, ele clica no botão *pedir*, navega para a tela de pedido, preenche as informações de entrega e finaliza o pedido.

Vamos criar a nova tela de pedido então.

A rota

Como vamos implementar um pedido simples de apenas um produto, vamos fazer a rota no formato `/pedido/:boloId` para indicar qual bolo estamos pedindo, parecida com a anterior de detalhes.

Acrescente uma nova rota no `js/routes.js` :

```
.state('pedido', {
  url: '/pedido/:boloId',
  templateUrl: 'templates/pedido.html',
  controller: 'PedidoController'
})
```

O controller

A princípio, o `PedidoController` vai fazer apenas a lógica de carregar qual é o bolo do pedido, igual ao que fizemos no `DetalheController` . Além disso, ele conterá um objeto `$scope.dados` para servir de modelo para o preenchimento do formulário de entrega.

Adicione o novo controller no final do arquivo `js/controllers.js` :

```
angular.module('starter')
.controller('PedidoController',
  function($scope, ProdutosService, $stateParams, $http) {

    ProdutosService.lista().then(function(dados){
        $scope.bolo = dados[$stateParams.boloId];
    });

    $scope.dados = {};
});
```

A view

A tela vai mostrar um resumo do pedido com nome e preço do

bolo, e exibir 3 campos: endereço, telefone e nome. Por fim, terá um botão para finalizar o pedido.

Crie no arquivo `templates/pedido.html` :

```
<ion-view>
<ion-nav-title>Seu pedido</ion-nav-title>
<ion-content>
    <div class="list list-inset">
        <div class="item item-divider">Bolo escolhido</div>
        <div class="item">{{bolo.nome}} por R${{bolo.preco}}</div>
    </div>

    <div class="list list-inset">
        <div class="item item-divider">Dados para entrega</div>

        <label class="item item-input">
            <input type="text" placeholder="Seu nome"
                   ng-model="dados.nome">
        </label>
        <label class="item item-input">
            <input type="tel" placeholder="Telefone"
                   ng-model="dados.telefone">
        </label>
        <label class="item item-input">
            <textarea placeholder="Endereço"
                      ng-model="dados.endereco"></textarea>
        </label>

        <button class="button button-positive button-full"
                ng-click="fecharPedido()">
            Confirmar Pedido!
        </button>
    </div>
</ion-content>
</ion-view>
```

Repare que cada um dos `<input>` possui um `ng-model` apontando para uma propriedade específica no objeto `dados` que criamos antes no controller.

Note também que o `<button>` que criamos possui um `ng-click="fecharPedido()"` . A ideia é ser um método no controller que vai finalizar o pedido. Ainda não o fizemos.

Chamada Ajax

O novo método `fecharPedido` deve ficar dentro do `PedidoController`. Seu papel é chamar a mesma URL que usamos antes na `garconapp` para registrar novos pedidos, a `http://cozinhapp.sergiolopes.org/novo-pedido`.

Essa URL agora recebe 2 parâmetros: `pedido` com o produto escolhido, e `info` com dados de entrega. Vamos montar os dois parâmetros a partir dos dados disponíveis no controller.

Acrescente dentro do `PedidoController`:

```
$scope.fecharPedido = function() {
    $http.get('http://cozinhapp.sergiolopes.org/novo-pedido', {
        params: {
            pedido: $scope.bolo.nome,
            info: $scope.dados.nome
                  + ' (' + $scope.dados.telefone + ') - '
                  + $scope.dados.endereco
        }
    });
};
```

Note que estamos usando o `$http` novamente. Ele precisa estar como parâmetro na função que define o controller.

Abra no navegador e teste fazer um novo pedido. Deixe em outra aba o painel de pedidos aberto para ver se seus dados chegam: http://cozinhapp.sergiolopes.org/.

O código está no GitHub:

https://github.com/sergiolopes/pedidapp/tree/472f.

17.3 INTERFACE MELHOR COM NOVOS COMPONENTES

Podemos fazer diversas melhorias em nossa App. Em particular, dar um feedback melhor para o usuário quando o pedido estiver sendo enviado.

Podemos usar um componente `$ionicLoading` para mostrar um *spinner* de carregando:

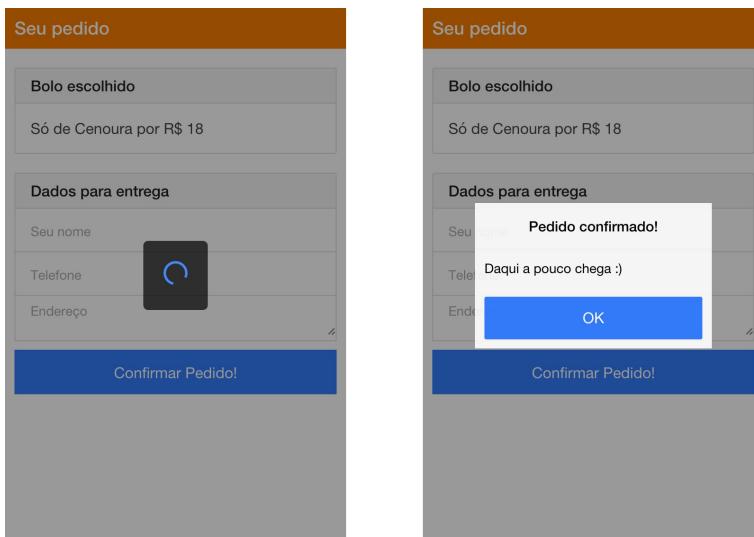
```
$ionicLoading.show();
```

Podemos também mostrar um pop-up de confirmação quando o pedido for registrado usando o `$ionicPopup` :

```
$ionicPopup.alert({
    title: 'Pedido confirmado!',
```

```
    template: 'Daqui a pouco chega :)'
});
```

E até navegar de volta para home após a confirmação do pedido com:

```
$state.go('home');
```

Juntando tudo isso e ainda fazendo um *alert* para o caso de o servidor devolver erro, chegamos a este novo `finalizarPedido()` mais robusto:

```
$scope.fecharPedido = function() {

    // mostra o spinner de loading
    $ionicLoading.show();

    // dispara a API
    $http.get('http://cozinhapp.sergiolopes.org/novo-pedido', {
        params: {
            pedido: $scope.bolo.nome,
            info: $scope.dados.nome
                  - ' (' + $scope.dados.telefone + ') - '
                  - $scope.dados.endereco
        }
    }).then(function() {

        // caso OK, mostra pop-up confirmando e
        // então navega pra home
        $ionicPopup.alert({
            title: 'Pedido confirmado!',
            template: 'Daqui a pouco chega :)'
        }).then(function(){
            $state.go('home');
        });

    }).catch(function(erro) {

        // caso dê erro mostra alerta com o erro
        $ionicPopup.alert({
            title: 'Erro no pedido!',
            template: erro.data + '. Liga pra gente: 011-1406'
        });

    }).finally(function(){
        // em qualquer caso, remove o spinner de loading
        $ionicLoading.hide();
```

```
    });
};
```

Um ponto importante é que, para poder chamar esses componentes `$ionicLoading` e `$ionicPopup`, além do `$state`, precisamos adicioná-los como parâmetros na definição do `PedidoController`. Algo assim:

```
angular.module('starter').controller('PedidoController', function(
$scope, $stateParams, $http, $state, $ionicPopup, $ionicLoading, P
rodutosService) {
```

O código está no GitHub em:

https://github.com/sergiolopes/pedidapp/tree/acad.

17.4 CONFIGURAÇÕES DO CORDOVA

Nossa App está pronta para rodar nos dispositivos reais. Mas podemos fazer alguns outros ajustes importantes que aprendemos.

Ícones e Splash Screens

No capítulo 12, vimos a importância do ícone e da splash screen na App, e a dificuldade em gerar todos os tamanhos de arquivos necessários. Inclusive, usamos o Ionic na ocasião para resolver isso. Faremos novamente.

Baixe os arquivos `icon.png` e `splash.png` daqui: https://github.com/sergiolopes/garconapp/tree/config/resources. Coloque ambos em uma nova pasta `resources` na raiz do nosso projeto `pedidapp`.

Agora execute no terminal:

```
ionic resources
```

Ele vai gerar todos os tamanhos possíveis no Android e no iOS, além de atualizar o `config.xml`.

Se preferir baixar no GitHub já com os arquivos gerados, acesse:

https://github.com/sergiolopes/pedidapp/tree/1061.

StatusBar

Havíamos usado o plugin de StatusBar em uma App anterior com Cordova. O Ionic já adiciona esse plugin por padrão quando criamos o projeto com ele. Então, podemos adicionar no `config.xml` as configurações para controlar a cor da barra como fizemos antes no capítulo 13:

```
<platform name="android">
    <preference name="StatusBarBackgroundColor" value="#E86C13" />
</platform>

<platform name="ios">
    <preference name="StatusBarBackgroundColor" value="#F57F17" />
</platform>
```

Podemos aproveitar também e configurar a orientação da App:

```
<preference name="Orientation" value="portrait" />
```

Lembre-se de que, no fim, um projeto Ionic é um projeto Cordova. Então, tudo o que vimos antes continua valendo.

O projeto final do capítulo está em:

https://github.com/sergiolopes/pedidapp/tree/0066.

CAPÍTULO 18

CACHE E OFFLINE

No quesito suporte a offline, aplicações híbridas têm já uma grande vantagem com relação a Web: todos sua estrutura, seu HTML, CSS e JS, são baixados durante a instalação. Isso quer dizer que, se fizermos uma App estática que já possui tudo necessário no código-fonte, **ela funciona offline automaticamente**.

A grande questão é quando envolvemos dados externos, que exigem chamadas Ajax, para serem obtidos. Como é o caso da nossa *pedidapp* que fizemos antes.

Toda a estrutura da App está disponível offline, mas os dados com a lista de produtos a serem vendidos vem do servidor. Se o usuário estiver sem internet ou com o sinal ruim, não conseguirá usar a App.

18.1 ESTRATÉGIAS PARA DADOS OFFLINE

A solução para se trabalhar offline é **persistir esses dados dinâmicos localmente no aparelho**. A ideia é que, se o usuário estiver offline, possamos usar os dados locais e oferecer uma boa experiência para o usuário.

Claro que na primeira vez o usuário vai precisar estar online para baixar os dados (assim como ele precisa estar online para instalar a App, obviamente). Mas os usos seguintes poderão fazer uso desse cache local.

Um ponto a se considerar é que os dados salvos localmente **podem não ser a última versão disponível no servidor**. Em muitas situações, isso é aceitável. Mas existem casos onde não podemos em hipótese alguma trabalhar com dados velhos. Aí não temos como suportar uso offline.

No caso das Apps da *Só de Cenoura*, nenhum dado é crítico. Todos podem ser cacheados localmente e não há problema se estiverem um pouco desatualizados.

Ainda assim, existem várias formas de trabalhar com offline. Devemos acessar o cache local apenas se a rede estiver indisponível? Devemos sempre devolver o cache local primeiro? Existem vantagens e desvantagens em cada cenário, e depende do caso. Vejamos.

Network-first com cache de fallback

Podemos sempre tentar **acessar a rede primeiro**. Só usamos o cache se houver algum problema. Então disparamos o Ajax e, depois de um tempo sem resposta, pegamos o dado do cache para usar. A ideia é configurar um *timeout* para o request para identificar se o Ajax falhou.

Até existem formas de saber se o dispositivo está sem rede ou não, mas são pouco úteis. Só vão detectar se a pessoa colocou modo avião ou se está completamente sem sinal. No mundo real, o sinal é variável e depende da qualidade de recepção. Além disso, pode ser que o problema não esteja no sinal do usuário, mas sim no nosso servidor - pode ter caído, ou pode estar com erro.

Vamos preferir a abordagem do *timeout* então. O problema disso, claro, é que. no pior caso. o usuário fica um tempo esperando o Ajax desistir para só aí pegar os dados do cache. É ruim para a sensação de performance.

Offline-first com Ajax para atualizar

Podemos inverter o pensamento. Sempre pegamos o dado local do **cache primeiro** e já mostramos esse dado para o usuário. Só depois disparamos um **Ajax em background** para verificar se temos dados mais novos no servidor.

A App do Facebook faz exatamente isso. Você abre e vê sua timeline instantaneamente com os dados da última vez que sincronizou, e ela logo começa a pegar os posts mais novos e atualiza a tela e a timeline para você automaticamente.

A vantagem dessa estratégia é que o usuário carrega a App instantaneamente. **A sensação de performance é ótima.** É ideal para casos onde ver os dados antigos primeiro não é tão grave.

Podemos pensar ainda em mais um detalhe: a atualização em Ajax que vai ser disparada em background deve atualizar a tela no momento que retornar? Ou podemos apenas salvar esses dados para a próxima vez que usarmos a App? Depende da importância desses dados atualizados para o usuário.

O Facebook opta por atualizar a timeline no momento que sincroniza. Mas pense, por exemplo, em como funciona o update do Chrome no Desktop: ele baixa a versão mais nova em background, mas só atualiza mesmo na próxima vez que você iniciar o navegador. Isso porque o update não é absurdamente importante e para não incomodar o usuário com a atualização.

No fim, precisamos pensar em cada caso e decidir qual a melhor estratégia.

18.2 SALVANDO DADOS NO DISPOSITIVO

Antes de implementarmos a estratégia de offline propriamente

dita, precisamos pensar em como salvar os dados no dispositivo. Existem algumas opções.

Web Storage API

Você pode usar o bom e velho `localStorage` para salvar dados localmente com Cordova. É uma API simples, onde você guarda chave/valor e depois recupera esses dados.

A vantagem é que funciona em todo tipo de navegador e WebView moderno. A desvantagem é que é limitado. Você só pode gravar strings, a API é síncrona, o que pode causar gargalos de performance, e o volume de dados salvos é limitado (depende do navegador, mas é da ordem de poucos megabytes).

Em muitos casos, como da *pedidapp*, o `localStorage` é suficiente. Para usá-lo, não precisamos de plugin, apenas chamar no JavaScript:

```
localStorage.setItem('chave', 'valor')
var valor = localStorage.getItem('chave')
localStorage.removeItem('chave')
```

IndexedDB

Existe na Web uma API melhorada para storage, que é a IndexedDB. Ela é assíncrona, transacional e suporta todo tipo de dado. Mais robusta que o `localStorage` , mas um pouco mais complicada de se usar.

A grande desvantagem é que não tem suporte universal. Em particular, as WebViews clássicas do iOS não suportam. Logo, na prática, não é muito útil em aplicações Cordova multiplataforma.

WebSQL

É possível usar um banco de dados local mais robusto na Web

com suporte a SQL inclusive. A API do WebSQL permite isso. O problema é que ela foi descontinuada, então seu futuro é incerto. O curioso é que, mesmo assim, boa parte dos navegadores implementam - a única exceção é Windows Phone / Internet Explorer.

Mesmo assim, a WebSQL nativa do navegador / WebView tem limitações de espaço total em uso. Se quiser trabalhar com banco local, veja a próxima opção.

Sqlite com plugin

As soluções anteriores de storage exploravam as disponíveis na Web e nas WebViews por padrão. Mas estamos no Cordova, e podemos usar plugins para usar recursos **nativos**.

Em particular, é bastante comum o uso do banco **sqlite** nas Apps Mobile. Muitas plataformas já disponibilizam como parte do sistema operacional, inclusive. Basta criar um banco e usar.

Podemos usar o plugin `cordova-sqlite-storage` para isso:

https://github.com/litehelpers/Cordova-sqlite-storage.

Ele funciona em muitas plataformas e oferece uma API parecida com a WebSQL. Para usá-lo, precisamos instalar o plugin:

```
cordova plugin add cordova-sqlite-storage --save
```

Um exemplo de código JavaScript que cria uma tabela e faz um `INSERT` em uma transação seria:

```
// abre o banco de dados
var db = window.sqlitePlugin.openDatabase({name: "app.db"});

// inicia transação
db.transaction(function(tx) {
    // cria tabela
    tx.executeSql(
```

```
        'CREATE TABLE Produtos (id integer primary key, nome text)'
    );

    // insere uma linha
    tx.executeSql('INSERT INTO Produtos (nome) VALUES (?)',
        ['Bolo de Nutella']);
});
```

Você pode consultar a documentação do plugin para aprender mais sobre ele.

Acesso a arquivos locais com plugins

Outra opção é o acesso direto aos arquivos do dispositivo. Isso pode ser útil para gravar arquivos grandes offline, como um cache de imagens ou músicas.

Usamos este plugin: https://github.com/apache/cordova-plugin-file/.

18.3 SOLUÇÃO OFFLINE-FIRST

Pesando todas as possibilidades que discutimos antes, parece fazer sentido na *pedidapp* uma solução Offline-first. Vamos sempre usar a versão do cache, se disponível, e só depois buscar na rede o dado mais atual. Ainda mais: como os dados não são de extrema importância, a versão atualizada só estará disponível na próxima vez que a App for aberta.

Além disso, vamos usar `localStorage` para guardar os dados, visto que temos poucas informações e esse é o jeito mais simples e portável existente. Ele é o mais limitado e, em algumas situações, pode até ser apagado pelo sistema. Mas como vamos usá-lo para cache, é suficiente.

Nossa implementação ficará encapsulada na `ProdutosService`, que já havíamos criado no `js/services.js`.

A ideia é que todo o restante da App não precisará ser alterado. Agora, a implementação está assim:

```
angular.module('starter')
.service('ProdutosService', function($http, $q) {

    var url = 'http://cozinhapp.sergiolopes.org/produtos';

    return {
        lista: function(){
            return $http.get(url).then(function(response){
                return response.data;
            });
        }
    };

});
```

Ou seja, cada vez que o método `lista()` é chamado, nós chamamos um novo Ajax e devolvemos uma *promise* que resolverá quando o Ajax retornar.

Uma primeira coisa que já podemos fazer para melhorar a performance é isolar a *promise* fora do `lista()` para que seja chamado apenas uma vez por execução da App.

```
angular.module('starter')
.service('ProdutosService', function($http, $q) {

    var url = 'http://cozinhapp.sergiolopes.org/produtos';

    var promise = $http.get(url).then(function(response){
        return response.data;
    });

    return {
        lista: function() {
            return promise;
        }
    };
});
```

O próximo passo é salvar no `localStorage` o retorno do serviço. Lembre de transformar o objeto em string:

```
var promise = $http.get(url).then(function(response){
    var json = JSON.stringify(response.data);
    localStorage.setItem('cache', json);
    return response.data;
});
```

Agora o que precisamos é verificar se existe um valor em cache e usá-lo imediatamente no `lista()` .

```
var cache = localStorage.getItem('cache');

if (cache != null) {
    promise = $q(function(resolve, reject) {
        resolve(JSON.parse(cache));
    });
}
```

A API do `localStorage` é síncrona, mas como o método `lista` devolve uma *promise*, embrulhamos o valor do cache em uma *promise* que resolve na hora. Assim, ela pode substituir a *promise* do Ajax sem quebrar quem estava chamando o `lista()` .

O código final do `ProdutosService` fica então:

```
angular.module('starter')
.service('ProdutosService', function($http, $q) {

    var url = 'http://cozinhapp.sergiolopes.org/produtos';

    // sempre dispara o serviço pra checar dados mais recentes
    var promise = $http.get(url).then(function(response){
        var json = JSON.stringify(response.data);
        localStorage.setItem('cache', json);
        return response.data;
    });

    // procura no localStorage
    var cache = localStorage.getItem('cache');
    if (cache != null) {
        promise = $q(function(resolve, reject) {
            resolve(JSON.parse(cache));
        });
    }

    return {
        lista: function() {
```

```
            return promise;
        }
    };

});
```

É um código peculiar. Vamos recapitular.

Quando o *service* for criado, ele sempre vai disparar um Ajax para pegar dados mais recentes, mas isso é assíncrono e demorará. Então, logo consultamos o `localStorage` para ver se temos dados locais. Se tivermos, disponibilizamos imediatamente esses dados - *offline first*.

Enquanto isso, alguma hora o Ajax vai retornar, e então salvará os dados novos no `localStorage` . Mas repare que esses dados não são usados agora. Optamos por aquela estratégia *a la Google Chrome*: os dados novos só vão ser usados na próxima vez que abrirmos a App.

É importante perceber também que, caso não existam dados no `localStorage` , a *promise* do Ajax vai ser devolvida. Ou seja, os dados que serão usados virão do Ajax - e vão demorar um pouco mais, mas só na primeira vez.

Por fim, vale lembrar que implementamos uma solução simples para um problema relativamente simples. Temos apenas uma chamada de API e ela devolve apenas um dado. De propósito, deixei as imagens em *data URIs* para serem salvas todas de uma vez. Em um projeto maior, talvez você precise usar uma solução mais complexa, como gravar o JSON no `localStorage` , mas as imagens com o plugin de acesso a arquivos.

Testando

Teste novamente e repare como, a partir da segunda vez, os dados vêm instantaneamente. Offline-first! Você pode desligar a

Internet e os dados ainda aparecem.

Lembre-se de que ainda estamos chamando o Ajax toda vez, **procurando por atualizações**. Como o serviço é estático, você não deve perceber isso. Então, implementei um truque: passe `?random=1` na URL da chamada que ele mistura o array na hora de devolver (e acrescenta um pouco mais de dados para ficar mais bagunçado também). Só trocar na chamada do Ajax:

```
http://cozinhapp.sergiolopes.org/produtos?random=1
```

Teste abrindo e fechando a App várias vezes. Repare como temos dados "novos" a cada vez, e de forma instantânea, vindo do cache primeiro.

O código completo desse exemplo você encontra em:

https://github.com/sergiolopes/pedidapp/tree/84f5.

18.4 OUTRAS IMPLEMENTAÇÕES DE OFFLINE

E se quiséssemos implementar as outras estratégias de *offline* que discutimos? Eu gosto bastante da solução que usamos da *pedidapp* de *Offline-first* com dados novos apenas na próxima execução.

Mas e se tivermos um cenário diferente?

Offline-first com dados novos

Todo o mecanismo de *offline-first* que fizemos continua valendo. Precisamos apenas adicionar uma forma nova de avisar os controllers que os dados novos chegaram. Há vários jeitos de fazer isso. Eu gosto de **eventos**.

A ideia é o service emitir um evento de aviso que dados novos

chegaram e os controllers interessados vão escutar esse evento. Com Angular, é bem fácil fazer isso.

No `ProdutosService`, quando os dados chegarem no Ajax (dentro do `then`), nós emitimos um evento global. Para isso, precisamos do `$rootScope`, que tem um método `$broadcast`, para enviar qualquer evento:

```
$rootScope.$broadcast('produtos-atualizados', response.data);
```

Em todos os controllers interessados nos dados novos, basta escutar esse evento com `$scope.$on` e passar o callback a ser executado. Por exemplo, podemos colocar no `HomeController`:

```
$scope.$on('produtos-atualizados', function(event, dados) {
    $scope.bolos = dados;
});
```

Teste e veja que os dados do cache são carregados na hora, e poucos instantes depois novos dados os substituem. Melhor ainda seria mostrar algum indicativo visual para o usuário de que os dados mudaram.

Network-first

E se precisarmos implementar a solução que vai na rede primeiro e só em caso de problema vai no cache?

Primeiro, adicionamos um `timeout` ao nosso Ajax, por exemplo, 5s. Se estourar ou der problema no servidor, a *promise* vai falhar e podemos capturar o erro em um bloco `catch`, e então fazer a leitura do `localStorage`. Ficaria algo como:

```
var promise = $http.get(url, { timeout: 5000 })
    .then(function(response){
        var json = JSON.stringify(response.data);
        localStorage.setItem('cache', json);
        return response.data;
    })
    .catch(function() {
```

```
    return JSON.parse(localStorage.getItem('cache'));
});
```

18.5 INDO ALÉM NO OFFLINE

Somente neste cenário dá para melhorar mais coisas, como atualizar os dados de tempos em tempos, tentar novamente o Ajax se falhar na primeira vez, ou pensar no que fazer quando nem rede nem cache estão disponíveis. Mas é preciso pensar nos outros pontos da aplicação também, por exemplo, a tela que registra o pedido. O que fazer se estivermos sem rede? Faz sentido pedido offline? Talvez não. Então, essa é uma parte da App que precisa de rede e pronto. Talvez seja bom melhorar a comunicação com o usuário sobre isso.

Em outras Apps, talvez faça sentido enviar os dados mais tarde quando a rede voltar. O pedido do bolo não faz sentido, se não o cliente pode acabar registrando um pedido no dia errado.

Mas um cliente de e-mail pode deixar para enviar as mensagens pendentes quando a rede voltar. Basta salvar a mensagem pendente e tentar enviar novamente mais tarde. Note que, para implementar *background sync* de verdade, com a App fechada, é preciso fazer em código nativo.

Uma melhoria possível na nossa tela de registro do pedido é salvar os dados do cliente para uso futuro. Assim, na próxima vez que ele fizer um pedido, já preenchemos os dados pessoais dele.

Por fim, é importante citar que, em cenários mais complexos, o maior problema de suporte offline acaba sendo a **sincronização consistente dos dados**. Imagine o usuário começar a editar alguma coisa offline e quando a rede volta, descobre que outro usuário já alterou aquele dado. Como resolver o conflito? Esse é um problema bastante complexo, que mereceria quase um livro sobre o assunto.

Na prática, existem frameworks para trabalhar com essa sincronização. O *Meteor* é um deles, porém há vários.

CAPÍTULO 19

TESTES, DEBUG E COMPATIBILIDADE

Boa parte do dia a dia do desenvolvedor é testando e debugando seu código. Testar em dispositivos reais é essencial para pegar problemas de performance e de compatibilidade.

Como estamos usando WebViews com tecnologias Web, temos as questões clássicas de portabilidade entre navegadores. Uns suportam certos recursos e outros não, mas existem uns detalhes aí no meio que precisamos discutir.

19.1 TIPOS DE WEBVIEW

A história do iOS

No iOS, as WebViews são baseadas no **WebKit**, a engine open-source por trás do Safari. São 99% idênticas ao Safari, mas possuem algumas diferenças. Até o iOS 7, havia a WebView clássica chamada **UIWebView**, que tinha diferenças de performance com relação ao Safari normal.

No iOS 8, a Apple lançou uma nova WebView, chamada **WKWebView**, além de manter a antiga ainda disponível. A nova é bem melhor e muito mais rápida, mas tinha uns bugs que impediam o Cordova de usá-la, até o iOS 9 finalmente corrigir.

Hoje é possível optar por qual WebView usar pelos plugins do Cordova. Por padrão, ainda é usada a *UIWebView*, mesmo no iOS 9.

Um ponto importante é que a WebView é parte do sistema da Apple. Isso quer dizer que só temos atualizações quando há atualizações do iOS como um todo.

A história do Android

O Android quando surgiu tinha como base o **WebKit** também. Tanto o navegador padrão quanto a WebView seguiam o WebKit, mas, geralmente, com muitas diferenças. Para piorar, como o sistema é aberto, muitos fabricantes alteravam o código do WebKit em seus dispositivos.

O resultado é um caos de dezenas de versões diferentes do WebKit rodando, e cada uma com seus bugs e diferenças de compatibilidade. Por exemplo, as WebViews da Samsung no Android 4.x não tinham suporte a *IndexedDB*; todos os outros tinham.

No Android 4.4, o Google melhorou o cenário migrando a WebView do WebKit para o **Chromium**, que é a base do Chrome. Muitas coisas melhoraram, com mais suporte e mais performance. Os fabricantes ainda fazem suas modificações, mas em menor grau.

A partir do Android 5.0, melhorou ainda mais. A WebView passou a ser **um componente independente do sistema operacional** e ganha atualizações constantes pela Play Store. Isso quer dizer que o Google controla a WebView e que os updates são frequentes, sem depender dos fabricantes atualizarem o sistema todo.

Temos, então, um cenário bastante positivo nos Androids recentes: uma WebView muito boa baseada no Chromium com

atualização constante. Mas e se nossa App precisa rodar nos dispositivos anteriores ao 4.4? O Android WebKit de antigamente era bem ruim e pode trazer vários problemas de compatibilidade.

Para resolver isso, surgiu o projeto **Crosswalk** com uma ideia brilhante. Eles implementaram uma WebView baseada no Chromium independente do Android e montaram um plugin que empacota essa WebView junto com sua App Cordova. Isso quer dizer que podemos distribuir, junto com a App, uma WebView moderna e previsível para todos os usuários, sem mais problemas de compatibilidade ou dores de cabeça com modificações de fabricantes.

O ponto negativo, porém, é que o **Crosswalk embute um Chromium inteiro na sua App**. Sua App salta de alguns megabytes de tamanho para várias dezenas de megabytes - só o Crosswalk costuma adicionar uns 20 MB. Isso pode ser ruim para não só o tamanho do download, como o uso de memória, já que um Chromium inteiro vai ser executado apenas para sua App.

Vamos ver como usar o Crosswalk. Cabe a você decidir em quais projetos e situações seu uso é justificável.

19.2 USANDO CROSSWALK NO ANDROID

O Crosswalk (https://crosswalk-project.org) vai embutir uma WebView completa, baseada no Chromium, junto com sua App Cordova. Usá-lo hoje é bem simples, basta adicionar um plugin.

```
cordova plugin add cordova-plugin-crosswalk-webview --save
```

Para buildar os `apk` s, execute:

```
cordova build android
```

Repare que ele gera 2 `apk` s diferentes, já que o binário do

Chromium muda de acordo com a arquitetura - `android-x86-debug.apk` e `android-armv7-debug.apk`. Você pode executar o projeto como fizemos antes:

```
cordova run android
```

Crosswalk apenas nos Androids velhos

Embutir o Crosswalk é interessante porque acaba com nossos problemas de compatibilidade, mas, por outro lado, acrescenta um peso a mais na App. Peso esse que pode ser desnecessário nos Android mais recentes que já são baseados em Chromium.

É possível, então, gerar `apk` s diferentes para versões diferentes do Android, de forma a usar o Crosswalk no Android 4.x, mas usar a WebView nativa no Android 5+. Depois na hora de publicar na loja, podemos subir os 2 `apk` s e indicar qual Android deve baixar qual `apk`.

Depois de adicionar o plugin do Crosswalk, podemos gerar o `apk` final incluindo o Crosswalk com:

```
cordova build --release
```

Para gerar o outro APK para Android 5+, precisamos remover o plugin e buildar novamente, mas indicando que essa versão é apenas para Android 5+ (SDK 21):

```
cordova plugin rm cordova-plugin-crosswalk-webview
cordova build --release -- --minSdkVersion=21
```

Na prática, você pode criar um script que automatiza essa geração - adicionando e removendo o Crosswalk e buildando as versões diferentes do `apk`.

19.3 USANDO WKWEBVIEW NO IOS 9

O Cordova ainda usa a *UIWebView*, mesmo no iOS 9. Isso porque há algumas pequenas diferenças com relação a *WKWebView*, e nem todas as Apps vão funcionar de cara.

Mas podemos optar explicitamente pela *WKWebView* a partir do iOS 9, e ter uma performance melhor. Basta usar o plugin `cordova-plugin-wkwebview-engine`. O importante é ter certeza de que o `cordova-ios` é da versão 4 ou superior. Rode `cordova platform` para conferir a versão.

Instale o plugin com:

```
cordova plugin add cordova-plugin-wkwebview-engine --save
```

E agora basta rodar:

```
cordova run ios
```

No iOS 9, ele vai usar a *WKWebView* e, no iOS 8, vai continuar usando a *UIWebView*.

19.4 DEBUG NO ANDROID

Como nossa App roda na WebView, é possível fazer **debug remoto** como se faz em um browser normal: usar o DevTools e todas suas ferramentas para debugar a App.

No Android, isso é possível em todas as WebViews baseadas no Chromium. Para debugar em qualquer sistema operacional:

1. No dispositivo, habilite a `Depuração USB` nas configurações de desenvolvedor;
2. Plugue o cabo USB no aparelho e no Desktop;
3. No Desktop, abra o Google Chrome e navegue para `chrome://inspect`;
4. Sua WebView deverá aparecer listada aí. Basta clicar em *Inspect*.

O DevTools to Chrome é bastante completo e você pode acessar vários recursos importantes, desde ver erros de JavaScript no *Console*, até inspecionar os elementos HTML e seu estilo, além de visualizar as conexões de rede e fazer *profilings* avançados de memória e renderização.

Para mais informações, consulte a documentação do Chrome:

https://developers.google.com/web/tools/chrome-devtools/debug/remote-debugging/remote-debugging.

19.5 DEBUG NO IOS

No iOS, o processo é semelhante, mas envolvendo o Safari. Isso exige que você tenha um Mac a disposição.

1. No aparelho, vá às *Configurações* > *Safari* > *Avançado*. Lá, habilite a opção **Inspetor Web**.
2. No Desktop, abra o Safari, vá às *Preferências*, aba *Avançado* e marque a opção *Mostrar menu Desenvolvedor*.
3. Conecte o cabo USB no aparelho e no Mac.
4. No Safari Desktop, vá ao menu *Desenvolvedor* e você deve ver seu dispositivo listado. Selecione a página para abrir o inspetor.

O debug no Safari possui muitas ferramentas também. Apesar de não ser tão completo quanto o do Chrome, já supre a maioria das necessidades.

Debug em outras plataformas

Para debugar no Windows Phone, em um Android antigo com WebKit, ou mesmo em um iOS se você não tiver Mac, a única opção é o **weinre**. Ele é bem ruim, limitado e cheio de bugs, mas quebra um galho. Melhor que nada.

A ideia é rodar um servidor local na sua máquina e acrescentar um script especial no seu HTML. Saiba mais em:

https://people.apache.org/~pmuellr/weinre-docs/latest/.

Capítulo 20
PUBLICAÇÃO NAS LOJAS

Depois de muito suor, chega o glorioso momento final: **publicar sua nova App nas lojas de aplicativos!** Grande momento, que é acompanhado de algumas complicações e burocracias.

Em primeiro lugar, você vai precisar de uma **conta de desenvolvedor** em cada uma das plataformas que for publicar, e isso é sempre pago. A Apple cobra 99 dólares anuais, o Google cobra 25 dólares uma vez, e a Microsoft 19 dólares anuais.

O processo de gerar o arquivo final para publicação varia em cada plataforma, mas essencialmente significa buildar a aplicação como já fizemos antes com Cordova, e depois **assinar digitalmente** essa App.

Um ponto importante é o ID da App que deve ser único e não pode ser mudado depois. Em geral, usamos um nome de pacote gerado a partir do domínio. Nos exemplos do livro, usei `org.sergiolopes.garconapp` . Use um ID único nos seus projetos.

De posse de uma App assinada e da sua conta de desenvolvedor, inicia-se o processo de **publicação**. Existe um trabalho meio marketeiro aí. Você precisa de um bom nome, uma boa descrição, um bom ícone e screenshots reais da sua App para mandar junto. Sem essas coisas, nenhuma loja aceita publicar.

Falando em aceitar a publicação, todas as lojas possuem algum

processo de aprovação. Google e Microsoft costumam ser tranquilos, com aprovação automatizada em pouco tempo.

A Apple é conhecidamente a mais chata de todas. Demora semanas e demanda a avaliação pessoal de algum funcionário de lá. Eles rejeitam Apps que não têm "qualidade suficiente", são ultraconservadores quanto a coisas que competem com eles e têm mil outras regras. É importante consultar as regras da Apple antes de começar a desenvolver sua App, para saber se sua ideia não tem chance de ser barrada depois. Mesmo assim, não há como conversar com eles; você ainda corre o risco de ter seu trabalho todo barrado de última hora.

Uma vez que a loja aprova sua App, ela fica disponível para seus usuários instalarem. Depois você segue o mesmo processo para atualizações - gera o pacote final, assina-o e sobe na loja para aprovação.

Em linhas gerais, é assim que funciona. Vejamos como fazer nas principais plataformas, Android e iOS.

20.1 PUBLICAÇÃO DA PLAY STORE

Criação da conta de Developer

Você precisa ter uma *Google Account*. Logado nela, vá a:

https://play.google.com/apps/publish/signup/.

Preencha o cadastro e pague a taxa de $25 dólares. É preciso um cartão de crédito internacional para isso.

Preparação da App

No terminal, gere o `apk` final com:

```
cordova build android --release
```

Talvez você receba um erro do tipo `MissingTranslation`. Nesse caso, crie um arquivo `platforms/android/build-extras.gradle` com o conteúdo a seguir, e exporte de novo:

```
android {
  lintOptions { disable 'MissingTranslation', 'ExtraTranslation' }
}
```

O `apk` será gerado em:

```
platforms/android/build/outputs/apk/android-release-unsigned.apk
```

Assinando a App

Se você nunca publicou nada, vai precisar de um par de chaves para assinar sua App digitalmente. Vamos gerar essas chaves primeiro.

Para isso, vamos usar o utilitário `keytool` no terminal. Ele veio instalado junto com o Java JDK, então deve estar disponível para você. Execute em um único comando:

```
keytool -genkey -v -keystore chave-android.keystore -alias alias_n
ame -keyalg RSA -keysize 2048 -validity 10000
```

Ele vai pedir uma senha para desbloquear a chave.

> **IMPORTANTÍSSIMO**
>
> Guarde essas chaves com sua vida. Você não pode perder esses arquivos; se perder não consegue mais atualizar a App. E se caírem em mão erradas, podem ser usadas para coisas maliciosas em seu nome.

Para assinar a App, usamos o `jarsigner`, que também veio no

JDK. Ele recebe o arquivo da chave e o arquivo `apk` que geramos antes. Execute em um único comando:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore c
have-android.keystore android-release-unsigned.apk alias_name
```

O último passo é rodar o utilitátio `zipalign` que vem no Android SDK. Ele ajeita algumas coisas no arquivo antes da publicação.

```
zipalign -v 4 android-release-unsigned.apk SoDeCenouraFinal.apk
```

Caso não esteja no seu PATH, o `zipalign` pode ser encontrado na pasta do Android SDK em `build-tools/VERSAO/zipalign`.

> É possível fazer a exportação e assinatura do `apk` pelo Android Studio se você o tiver aberto. A complexidade é a mesma, mas você faz com uma interface visual.
>
> Consulte: http://developer.android.com/intl/pt-br/tools/publishing/app-signing.html#studio.

Subindo a App na Play Store

Feita nossa conta de developer e gerado o `apk` assinado, podemos iniciar a publicação. Acesse: https://play.google.com/apps/publish/.

Crie uma nova App, escolha a língua, dê um nome e faça upload do APK. Em seguida, vá seguindo as instruções para incluir as outras informações. Ele pede descrição, screenshots, classificação etária e outros dados. Não coloquei no livro, pois essas opções mudam, mas é fácil de acompanhar pelo passo a passo na Store.

No final, o botão **Publicar App** fica disponível. Então, é só clicar. Depois de algumas horas, você já deve achar sua App na Store.

20.2 PUBLICAÇÃO NA APPLE STORE

Conta de Developer

Para fazer a publicação, você precisará da conta de Developer, que custa 99 dólares anuais. Para abrir sua conta developer, acesse:

https://developer.apple.com/programs/enroll/.

Você precisará de um cartão de crédito internacional para o pagamento.

Criação do App ID

Precisamos registrar o ID da nossa App. É o mesmo ID que você colocou no XML do Cordova, por exemplo, `org.sergiolopes.teste` . Acesse com sua conta developer:

https://developer.apple.com/account/ios/identifiers/.

Lá, crie um novo App ID. No *Bundle Identifier*, coloque o mesmo ID da sua App Cordova.

Geração das chaves

O primeiro passo é criar um *Certified Signing Request* (CSR).

1. No Mac, abra o aplicativo **Keychain Access**.
2. Vá ao menu *Keychain Access > Certificate Assistant > Request a Certificate from a Certificate Authority*.
3. Preencha seu nome, um nome para a chave e deixe o CA Email em branco. Selecione a opção *Save to disk*, e finalize a criação.

Agora precisamos criar os certificados para produção:

1. Acesse https://developer.apple.com/account/ios/certificate/.
2. Crie um novo certificado do tipo *Production, App Store*.
3. Faça upload do CSR que geramos antes.
4. Finalize a criação e faça o download do certificado.
5. Abra o arquivo no Mac e adicione na Keychain.

Agora precisamos criar um *provisioning profile* para produção:

1. Acesse https://developer.apple.com/account/ios/profile/.
2. Crie um novo profile do tipo *Distribution, App Store*.
3. Selecione o App ID que criamos antes.
4. Em seguida, selecione o certificado de produção que criamos no passo anterior.
5. Dê um nome para o profile e finalize. No final, faça o seu *Download*.
6. Abra o arquivo baixado.

Exportação da App assinada

O processo de exportação, assinatura e envio da App deve ser feito pelo Xcode, logo, abra o projeto no Xcode. Ele está em:

```
platforms/ios/Projeto.xcodeproj
```

1. Abra o projeto no Xcode.
2. No canto esquerdo superior, próximo ao botão *Play*, selecione *Generic iOS Device*.
3. Vá no menu *Product* > *Archive* para gerar o arquivo.
4. Após a geração do pacote, clique em *Validate*. Selecione o *provisioning profile* que criamos antes.
5. Agora, clique em *Upload to App Store* e selecione novamente seu *provisioning profile*.

Opcionalmente, você pode também exportar o arquivo

localmente para depois fazer o upload na loja.

BUILD SEM MAC

É possível usar o PhoneGap Build para gerar o pacote final assinado da sua App para produção. Basta fazer upload do certificado e do provisioning profile, gerar por lá e baixar o arquivo `ipa` final.

Publicando a App no iTunes Connect

Feita nossa conta de developer e gerado arquivo final, podemos iniciar a publicação pelo **iTunes Connect** em:

https://itunesconnect.apple.com.

Crie uma nova App e siga as instruções. Use o Bundle ID que você criou antes. Você vai precisar preencher muitas informações e subir screenshots. Acompanhe as instruções.

Você pode fazer o upload do arquivo `ipa` exportado no Xcode. Ou, se fez o upload direto pelo Xcode, seu build já deve estar disponível no iTunes Connect para selecionar.

No fim, você pode enviar sua App para **review**, um processo que demora algumas semanas. Aí então sua App estará disponível na loja.

CAPÍTULO 21

INDO ALÉM

Estamos chegando ao final do livro. O objetivo era entender como o Cordova funciona, onde o Phonegap se encaixa, como usar plugins nativos, e como escrever uma *single page application*. Agora, o céu é o limite.

Se você quiser continuar o projeto do *Só de Cenoura*, há muito espaço para novas funcionalidades. Algumas ideias são:

- Faça uma vibração quando o pedido for confirmado com o `cordova-plugin-vibration` .
- Use o `cordova-plugin-geolocation` para oferecer o preenchimento do endereço de entrega de forma automática.
- Expanda as características de offline da App. Salve localmente os dados preenchidos pelo usuário. Use o plugin de Sqlite em vez do `localStorage` .
- Crie uma forma do usuário ter login e senha na App.
- Implemente um carrinho de compras onde o usuário possa adicionar vários produtos antes de fechar o pedido.

Se estiver querendo desafios maiores, implemente *Push Notifications* na App para avisar o cliente quando o pedido dele estiver a caminho da entrega. Não é uma tarefa simples, mas é interessante. Isso exige mudanças no back-end para enviar a notificação. Além disso, há toda uma complexidade na App em se criar chaves específicas nos serviços de cada plataforma. Recomendo

usar o *Ionic Push*, que já resolve algumas questões.

O ponto é que, entendida a arquitetura do Cordova, seu funcionamento e suas limitações, agora é questão de apenas ir implementando novas funcionalidades. Boa parte das coisas são feitas em HTML, CSS e JavaScript simples, como na Web. E para todas as outras - como as notificações -, existem plugins que você pode instalar. Cada plugin tem uma forma de uso própria, basta consultar a documentação individualmente.

Obrigado por acompanhar o livro. Se você tiver dúvidas, pode me encontrar no fórum da editora Casa do Código, em http://forum.casadocodigo.com.br/.

O que você vai criar com tudo o que aprendeu aqui?